MASTER EN SCIENCES INFORMATIQUES

Étude comparative de macrogénérateurs

Pirotte, B.

Award date:
1974

Awarding institution:
Universite de Namur

FACULTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX - NAMUR

INSTITUT D'INFORMATIQUE

1973-1974

ETUDE COMPARATIVE DE MACROGENERATEURS

B. PIROTTE

Mémoire présenté en vue de l'obtention
du grade de Licencié et Maître en Informatique

Nous tenons à remercier vivement Monsieur C. Cherton pour l'attention et l'aide constantes qu'il nous a accordées pendant l'élaboration et la rédaction de ce travail.

Nous remercions également Monsieur H. Leroy pour l'intérêt qu'il a manifesté dans la mise au point de ce mémoire.

Nous exprimons notre reconnaissance au Dr. Whitby-Strevens ainsi qu'aux membres du "Department of Computer Science - University of Warwick - Coventry", pour l'accueil qu'ils nous ont réservé et le travail qu'ils nous ont permis de réaliser pendant notre stage.

Nous exprimons notre gratitude à Mademoiselle G. Charlier pour le visage qu'elle a donné à ce mémoire.

Nous tenons à exprimer nos remerciements à tous les membres du corps professoral de l'Institut d'Informatique pour l'enseignement qu'ils nous ont donné depuis trois ans.

Namur, le 15 juillet 1974.

S O M M A I R E

0. INTRODUCTION : QU'EST-CE QU'UN MACROGENERATEUR ?

0.1 Aperçu historique

0.2 Principe de base d'un macrogénérateur

0.3 Plan de travail

0.1 Aperçu historique

Dans l'histoire des langages de programmation, la technique de macrogénération est apparue très tôt, mais jusqu'aux années 65 elle s'est développée uniquement dans le cadre des langages d'assemblage.

Au départ, elle fut introduite comme une extension rudimentaire de ceux-ci (sous-routines ouvertes composées d'instructions-assembleur, insérées telles quelles - avec substitution des paramètres actuels - aux endroits d'appel).

En 1960, McIlroy (1) proposait l'élargissement de cette technique par l'adjonction de variables et directives de macrogénération (assemblage conditionnel, calcul d'expressions, itération sur des chaînes de caractères, ...). Cela revenait à ajouter aux langages d'assemblage des instructions analogues aux instructions les plus simples des langages évolués, avec la différence qu'elles sont exécutées at compile-time et non at run-time.

Aux environs de 1965, deux autres tendances se font jour :

1.- d'une part, l'introduction de macros dans les langages évolués. Les créateurs de NPL, le précurseur de PL/I, stipulaient l'emploi de macros parmi les "compile-time facilities" (2). Cette première tendance fut la source de nouvelles recherches en matière d'extension des langages évolués (possibilité laissée à l'utilisateur d'un langage de définir lui-même des nouvelles structures syntaxiques).

2.- d'autre part, la conception des premiers "general purpose macrogenerators". Les macro-assembleurs, le macroprocesseur de PL/I, le macro-algol sont toujours associés à un seul langage, leur "langage de base". D'où leur rassemblement sous la désignation de "special purpose macrogenerators". La libération de cette contrainte amena la création de langages plus généraux, basés uniquement sur les principes de macrogénération. Leur emploi

---

(1) McIlroy : "Macroinstruction extensions of compiler languages" 1960

(2) G. Rodin et H.P. Rogoway : "Highlights on a New Programming language" 1965.

Ces deux articles sont repris dans S. Rosen : "Programming Systems and Languages" McGraw Hill 1967.

est encore restreint, mais par leur généralité même, ils arrivent à concurrencer et même à supplanter les langages évolués dans certains domaines(extension des langages, édition de textes, ...).

Cet aperçu succinct montre que, bien que la technique de macro-génération ait été développée d'une manière fort arbitraire, elle est devenue assez universelle pour que nous l'examinions de plus près. Aucune approche théorique du problème n'ayant vu le jour, nous nous proposons de faire une étude comparative de quelques macrogénérateurs représentatifs, ceci pouvant éventuellement constituer l'amorce d'une étude plus globale.

0.2 Principe de base d'un macrogénérateur

Un macrogénérateur est une pièce de software qui produit, à partir de textes source qui lui sont fournis, des textes cible obtenus par modification du texte source, d'une FACON INDIQUEE DANS CELUI-CI.Ce qui caractérise les macrogénérateurs, c'est la technique utilisée pour indiquer ce qui doit être généré. Décrivons la brièvement.

La génération se fait à partir de certaines parties seulement du texte source (les macro-instructions ou appels) grâce à des clichés (les macrodéfinitions) fournis par d'autres parties du texte source. Celui-ci est constitué de chaînes de caractères de deux types :

1.- des macrodéfinitions, comportant chacune deux composantes :
   - la ligne prototype, qui spécifie la syntaxe des strings appelant cette définition
   - le texte de remplacement ou corps, qui spécifie ce qui doit être généré à la place de la macro-instruction appelante.

2.- des macro-instructions, parties du texte source qui appellent une macrodéfinition.

L'action du macrogénérateur consiste à scanner le texte source afin de reconnaître et mémoriser les macrodéfinitions, et de distinguer les appels. Pour chaque appel, il doit identifier la définition appelée, passer les arguments et évaluer le texte de remplacement.

0.3 Plan de travail

Dans un premier chapitre, nous présentons les mécanismes auxquels recourent sept macrogénérateurs représentatifs pour réaliser ce principe de base. L'étude comparative sera faite suivant quatre axes :

1.- l'environnement : ensemble des éléments que le macrogénérateur peut référencer à chaque instant. L'environnement contient la sémantique du macro-langage, c'est-à-dire les indications pouvant apparaître dans le texte source et les actions que leur évaluation entraîne.

2.- la syntaxe des macro-langages : reconnaissance des macro-instructions et des macro-définitions

3.- les mécanismes d'évaluation du texte source : action(s) que la reconnaissance d'une structure syntaxique entraîne (sémantique des textes de remplacement)

4.- l'assemblage des parties du texte généré .

Il nous semble que ces quatre axes permettent de cerner les traits fondamentaux des techniques de macrogénération.

L'écriture de quelques macros illustrera les caractéristiques principales des macrolangages examinés. Ceci fera l'objet du second chapitre.

Dans un dernier chapitre, nous tâcherons d'évaluer certains aspects des techniques de macrogénération. Cette tentative nous amènera à discerner d'une part certaines limitations à conserver ou à introduire, d'autre part certaines extensions déjà partiellement réalisées ou souhaitables. Pour ce faire, nous définirons trois termes (cfr. introduction du chapitre 3) qui seront parfois utilisées dans les deux chapitres précédents : puissance, souplesse et efficience.

En annexe, nous présentons brièvement l'implémentation d'un subset de STAGE2 que nous avons réalisée sur le MODULAR ONE au département d'Informatique de l'Université de Warwick à Coventry.

1. ETUDE COMPARATIVE DE SEPT MACROGENERATEURS

1.1 Environnement

1.1.A Environnement initial
Tableau 1 : environnement initial

1.1.B Parties statiques et dynamiques de l'environnement
1.1.B.1 Macrodéfinitions statiques $\Longrightarrow$ variables
1.1.B.2 Macrodéfinitions dynamiques

1.1.C Contenu instantané de l'environnement
1.1.C.1 Environnement global et local
1.1.C.2 Création et suppression des liaisons
1.1.C.3 Accès aux liaisons
Tableau 2.A. : environnement global
Tableau 2.B. : environnement local

1.2 Syntaxe des macrolangages

1.2.A Syntaxe des macro-instructions
1.2.A.1 Délimitation des macro-instructions
1.2.A.2 Identification de la macrodéfinition appelée :
- par nom - par nom distribué
1.2.A.3 Reconnaissance des arguments
1.2.A.3.1 XPOP : notation étendue
1.2.A.3.2 ML/I : structure de délimiteurs
1.2.A.3.3 LIMP et STAGE 2 : noms distribués
1.2.A.4 Nombre variable d'arguments

1.2.B Syntaxe des macrodéfinitions

1.3 Mécanismes d'évaluation

1.3.A Conversions de paramètres

1.3.B Macro-instructions

1.3.C Macrodéfinitions

1.3.D Variables de macrogénération

1.3.E Directives de macrogénération
1.3.E.1 Directives de calcul
1.3.E.2 Directives de transfert

1.3.F Inhibitions des mécanismes d'évaluation
Tableaux 4.A. et 4.B. : évaluation du texte

1.4 Macrogénération différée

Nous examinons deux "special purpose macrogenerators" : le MACRO-ASSEMBLER IBM OS/360, dont nous relevons les caractéristiques générales (les caractéristiques particulières variant d'une implémentation à l'autre, le lecteur est prié de recourir aux manuels de référence propres à chaque implémentation) et XPOP, dont nous retenons uniquement quelques aspects originaux. Le langage de base de ce dernier est l'assembler-FAP.

Les cinq autres - GPM, TRAC, LIMP, ML/I et STAGE2 - sont des "general purpose macrogenerators".

1.1 L'environnement

L'environnement contient à chaque instant toutes les indications de modification nécessaires pour analyser et évaluer le texte input. Pour interpréter celui-ci, il est donc indispensable que des conventions de base soient prédéfinies . A défaut de ceci, il serait insensé d'attendre du macrogénérateur qu'il identifie les indications fournies par le texte source, puisqu'il ne disposerait pas des moyens pour les reconnaître!

1.1.A Environnement initial

Avant d'aborder le texte source, le macrogénérateur dispose donc d'un certain nombre"d'entités" (marqueurs, macros-système,...) grâce auxquelles il peut analyser le texte. L'ensemble de ces entités constitue l'environnement initial, noyau auquel viendront s'ajouter d'autres entités (macros-utilisateur, arguments, ...) au cours de la macrogénération.

Le tableau I précise le contenu initial de l'environnement pour chaque macrogénérateur.

La distinction entre marqueurs, directives, macros-système, ... n'est qu'une question de terminologie. Ils ont en commun une caractéristique essentielle : la reconnaissance de chacun d'eux dans le texte source entraîne une action ou une suite d'actions qui lui sont spécifiques. A chaque marqueur,macrosystème, ... est donc associé une procédure. Si nous convenons que ces procédures appartiennent à l'environnement, nous concluons que l'environnement initial est constitué du macro-générateurs lui-même. Ceci n'étant qu'une question de convention, nous

ASS	marqueurs (= caractères réservés) directives (MACRO, MEND, AIF, LCLA, ...) fonctions intrinsèques ($SYSNDX, $SYSLIST, T', ...)
XPOP	marqueurs pseudo-instructions (MACRO, CHPUNC, WAIT, ...)
GPM	marqueurs macros-système - de définition et mise à jour de macrodéfinitions (DEF, UPDATE) - de copie littérale du corps d'une macrodéfinition (VAL) - de calcul arithmétique (DEC, BIN, BAR)
TRAC	marqueurs macros-système : - de définition et segmentation de strings (DS, SS) - de suppression de définitions (DD) - de calcul arithm. et booléen (AD,SU,BU,...) - de comparaison (EQ, GR) - de manipulation de strings (CS, CC, CN, IN, CR)
LIMP	conversions de paramètres, noms de strings et "pattern elements" code interne de NL (newline) deux variables globales (pour les deux marqueurs)
ML/I	macros-système - de définition (suppression) de macrodéfinitions: MCDEF (MCNODEF) - de définition (suppression) de'warning marker' : MCWARN (MCNOWARN) - de définition (suppression) d'inhibitions : MCSKIP (MCNOSKIP) - de définition (suppression) d'insertions : MCINS (MCNOINS) - de définition de 'constructions' globales : MCDEFG, MCWARNG, MCSKIPG, MCINSG - de calcul arithmétique : MCSET - de transfert : MCGO
STAGE2	conversions de paramètres et fonctions-processeur code interne de NL des variables globales (pour les marqueurs)

TABLEAU 1 : ENVIRONNEMENT INITIAL

indiquons uniquement les marqueurs, ... dans l'environnement initial. Les actions qu'ils entraînent seront examinées dans la suite du chapitre

1.1.B Parties statiques et dynamiques de l'environnement

Nous illustrerons l'examen des différentes parties de l'environnement au moyen de quelques schémas. Dans ce but, nous adoptons les conventions suivantes :

i , j	désignent des entiers positifs
~i	désigne le paramètre formel i
arg i nom i valeur i	représentent des strings (identifiés par l'indice i)
-----	représente un string quelconque
[nom i, -------] [nom i, valeur i]	sont des macrodéfinitions : la partie à gauche de la virgule représente la ligne prototype, la partie droite le texte de remplacement ou corps
(nom i, arg i, ..., arg n)	est une macro-instruction à n paramètres actuels qui appelle la macrodéfinition nom i

Les délimiteurs] et) doivent être balancés par rapport aux délimiteurs respectifs [et (.

↑	pointe vers l'endroit (du texte source) que le macrogénérateur évalue.
⇑	pointe vers l'endroit où le macrogénérateur doit reprendre le scanning après évaluation du corps de la macro appelée (voir ci-dessous).
string 1 ⟶ string 2	représente une liaison (ex.: identificateur ⟶ valeur)

Ces conventions sont assez générales, bien qu'elles ne tiennent pas compte de certains aspects particuliers (nombre variable d'arguments key words, ...). La prise en considération de ces aspects alourdirait inutilement la notation.

L'environnement initial contient les marqueurs :

,	séparateur
(	début de macro-instruction
)	terminateur de macro-instruction
[	début de macro-définition
]	terminateur de macro-définition

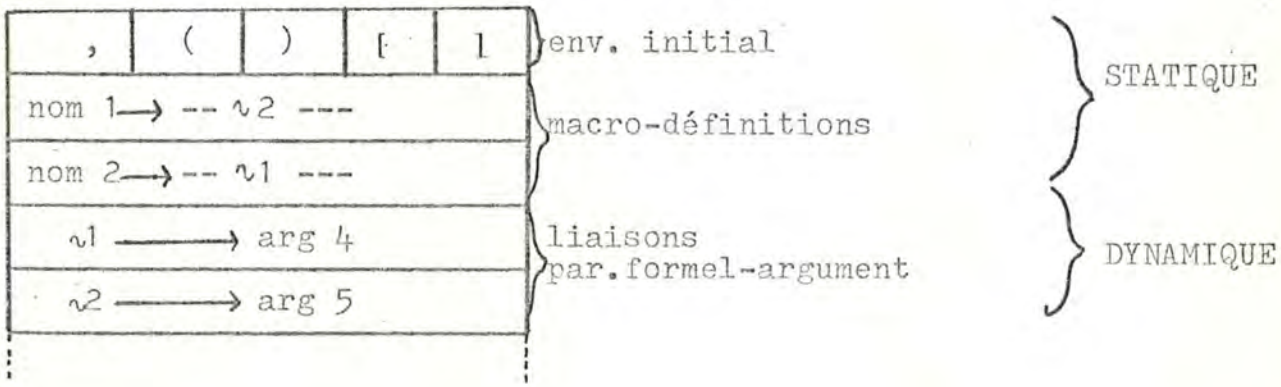
La reconnaissance d'une macro-définition[nom i, ----] entraîne son introduction dans l'environnement : celui-ci s'enrichit donc de la liaison "nom i⟶-----", ce qui permet,dès ce moment, d'appeler cette macro-définition.

Prenons comme hypothèse que toutes les macro-définitions sont dans l'environnement avant le début de la génération (la partie macro-définitions est statique). Lors de la rencontre d'une macro-instruction, le macrogénérateur interrompt le scanning du texte source (en mémorisant l'endroit où il s'interrompt par le pointeur ⇑) pour évaluer le corps de la macro-définition appelée : cette opération d'évaluation est la macro-expansion. Lorsque celle-ci est terminée, l'évaluation reprend à l'endroit pointé par ⇑ . Pour pouvoir référencer les arguments pendant la macro-expansion, le macrogénérateur doit donc avoir introduit dynamiquement les liaisons "paramètre formel⟶argument" dans l'environnement (le ième argument de l'appel est lié au paramètre formel ∿i).

Etant donné la situation suivante

[nom 1, ----- ∿2 ----][nom 2, ---- ∿1 ---] (nom 1, arg 4, arg 5) ---

l'environnement est



Si la partie dynamique concerne uniquement les liaisons"paramètre formel ⟶ argument", la puissance du macrogénérateur est faible : l'expansion d'une macro se réduit à une copie du corps avec substitution des paramètres actuels aux paramètres formels. Mais si le macrogénérateur peut à chaque instant tenir compte de certaines actions qu'il a accomplies antérieurement, la puissance augmente considérablement.
Tenir compte d'une action antérieure, c'est retrouver une information qui
- ou bien a été créée et mémorisée à la suite de cette action
- ou bien a été modifiée (et avait donc été créée avant) à la suite de cette action.

La possibilité d'introduire dynamiquement des liaisons "symbole ⟶ valeur" (appelées variables de macrogénération) dans l'environnement et de modifier dynamiquement la partie droite (valeur) de telles liaisons accomplit cela : la partie gauche identifie la partie droite, celle-ci constituant l'information. Comme les deux sont liées, il suffit de référencer le symbole pour pouvoir soit retrouver l'information, soit la modifier.
Afin de prendre en considération des actions antérieures, les dynamiques de création et de modification des variables doivent pouvoir être liées aussi bien à la macrogénération considérée comme un tout qu'à chaque macro-expansion.

La réalisation de ces mécanismes dépend uniquement des possibilités offertes pour la définition des macros. Deux cas sont à envisager

1.1.B.1 Macrodéfinitions statiques ⟶ variables

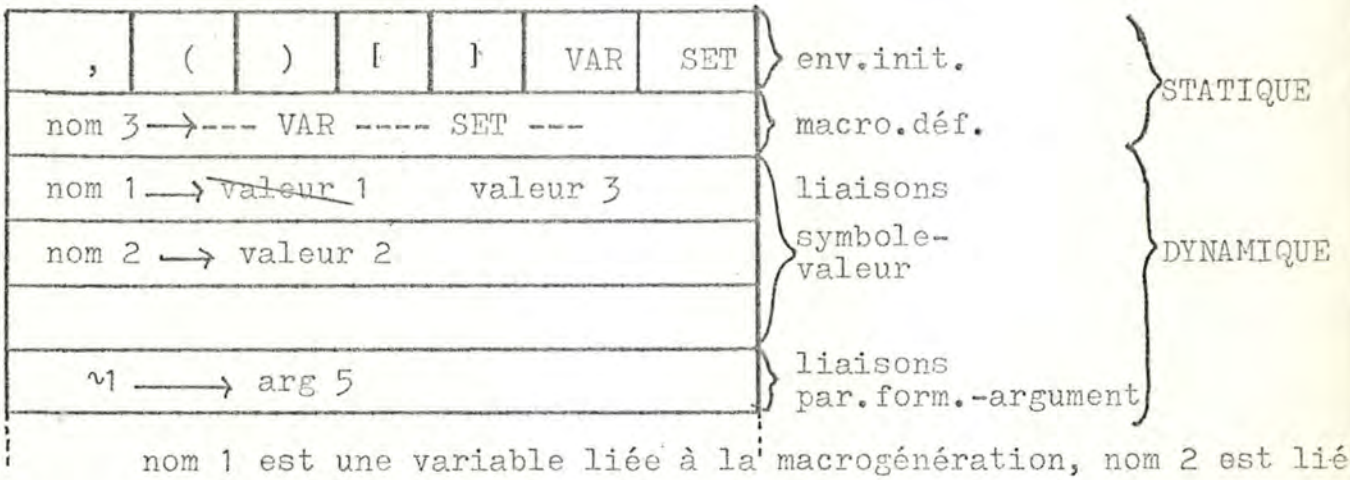
Le macrogénérateur dispose de toutes les macrodéfinitions avant de commencer la macrogénération proprement dite.
Il faut donc que l'environnement initial contienne des mécanismes permettant d'introduire des variables dans l'environnement et de les modifier. Nous ajoutons donc les conventions suivantes :

VAR, nom i, valeur i	crée une liaison "symbole ⟶ valeur"
nom i	délivre la valeur du symbole identifié
SET, nom i, valeur i	modifie la partie droite de la liaison référencée

la situation suivante :

VAR, nom 1, valeur 1 [nom 3, --- VAR, nom 2, valeur 2 --- SET, nom 1, valeur 3 ----] (nom 3, arg 5) ----

implique l'environnement



nom 1 est une variable liée à la macrogénération, nom 2 est liée à l'expansion de la macro nom 3.

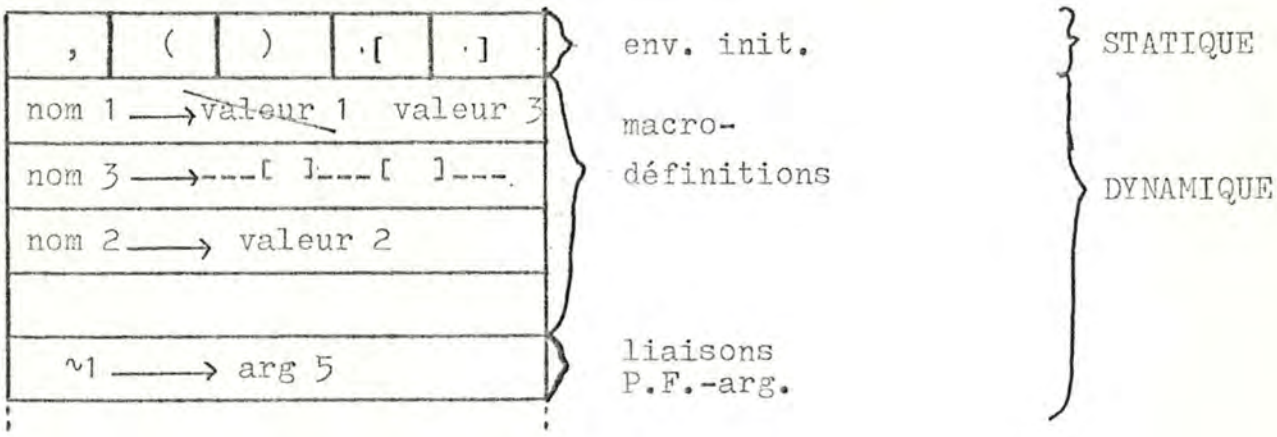
NOTE : Nous avons choisi deux directives différentes : une pour créer, une pour modifier les liaisons symbole-valeur. Une seule directive aurait suffi , à la condition de pouvoir modifier une liaison par redéfinition.

1.1.B.2 Macrodéfinitions dynamiques

Si nous pouvons définir et redéfinir des macros au cours de la génération, le mécanisme de macrodéfinition satisfait les conditions requises pour créer et modifier des liaisons "symbole ⟶ valeur". La valeur (corps) d'un symbole (ligne prototype) sera délivrée par l'appel de la macrodéfinition. Ceci suppose que la reconnaissance et l'évaluation des appels se fassent dynamiquement (ce qui est le cas de tous les macrogénérateurs étudiés).

Reprenons la situation précédente, cela donne :

[nom 1, valeur 1] [nom 3, --- [nom 2, valeur 2] ---[nom 1, valeur 3] ---] (nom 3, arg 5) ---



1.1.C Contenu instantané de l'environnement

Jusqu'ici, nous avons passé sous silence un aspect fondamental de l'environnement dynamique : la portée des liaisons qu'il contient. Ainsi, lorsqu'une macro-instruction est reconnue, le macrogénérateur doit introduire les arguments dans l'environnement avant d'entamer la macro-expansion. Celle-ci terminée, les arguments deviennent inutiles et disparaissent de l'environnement. Le temps de présence (portée) des arguments d'un appel dans l'environnement est donc la durée d'expansion de la macro appelée.

1.1.C.1 Environnement global et local

La portée des liaisons se subdivise en deux classes :

1.- globale : c'est la durée de macrogénération, excepté si une indication explicite réduit la portée à une partie de cette durée. En ce qui concerne les variables, cette portée suffit largement au transfert d'informations d'une macro-expansion à n'importe quelle expansion ultérieure.

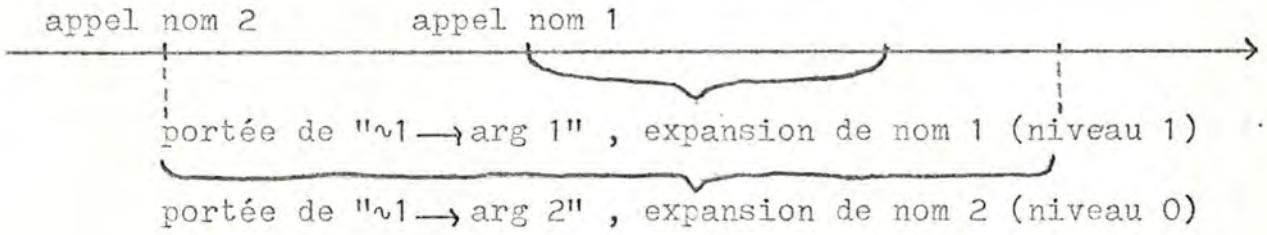
2.- locale : la portée des liaisons locales est la durée de macro-expansion, sauf si elle est explicitement restreinte. Elle suffit au travail interne à une expansion. Nous devons tenir compte ici des macro-instructions imbriquées dans les macrodéfinitions. Comme l'expansion des appels imbriqués est dynamique, nous distinguons des niveaux de macro-expansion: le niveau 0 est celui des appels extérieurs aux macrodéfinitions, et les appels générés par l'évaluation du corps d'une macro de niveau i sont de niveau i + 1. La portée des liai-

sons créées pendant l'expansion d'une macro de niveau i inclut les durées des expansions des macros imbriquées (niveau> i). Les liaisons locales sont donc mémorisées sur un stack : le début d'une macro-expansion introduit des liaisons locales au sommet de la pile, la fin supprime toutes les liaisons locales de ce niveau. Elles appartiennent donc toujours à un niveau, à une "couche" de l'environnement local

L'évaluation de (nom 2, arg 2) dans le texte suivant

[nom 1, ---] [nom 2, --- (nom 1, arg 1) --] (nom 2, arg 2)

suit donc le diagramme du temps



Avant d'aborder la question de l'accès aux liaisons, nous examinons comment les liaisons entrent et sortent de l'environnement.

1.1.C.2 Création et suppression des liaisons

Le critère de création-suppression n'est pas l'écriture du texte, mais son évaluation. Ainsi, les indications de création-suppression apparaissant dans un texte de remplacement ne sont pas opérationnelles lorsque la macrodéfinition est introduite dans l'environnement, mais lorsqu'elles sont évaluées suite à un appel. L'éxécution des indications suit la séquence d'évaluation, non la séquence textuelle !

Suivant les cas, les indications sont implicites ou explicites :

1.- Implicite : la création (suppression) des liaisons globales coïncide avec le début (fin) de macrogénération , la création (suppression) des liaisons locales avec le début (fin) de macro-expansion .

2.- Explicite : lorsque l'environnement initial contient des directives explicites de création (suppression), la création (suppression) des liaisons se fait lors de l'évaluation de ces directives.

L'examen des tableaux 2.A et 2.B montre que

- les créations sont soit implicites, soit explicites

- la création et la suppression des arguments sont presque toujours implicites, et que dans ce cas, les arguments ne peuvent pas être modifiés : ils appartiennent donc à un environnement local statique. Seuls LIMP et STAGE2 permettent de créer et de modifier des liaisons paramètre formel-argument en cours d'expansion : les arguments peuvent alors être utilisés comme variables locales.

- les suppressions sont habituellement implicites. Certains macro générateurs possèdent des ordres de suppression explicites, dont l'emploi est facultatif. A défaut de ces ordres, la suppression est implicite. En TRAC, il faut spécifier la ou les définitions à supprimer. En XPOP, LIMP et ML/I, l'emploi d'un ordre supprime toutes les liaisons - du niveau courant lorsqu'il s'agit d'ordres locaux - de la classe spécifiée (macrodéfinitions, variables,...)

- XPOP ne possède pas de liaisons dynamiques. Mais ce macrolangage possède une pseudo-instruction permettant d'assembler et d'exécuter des parties du texte source at macro-time. Ce mécanisme suffit pour permettre des calculs ou des transferts de contrôle en cours de macrogénération.

- TRAC ne possède pas de liaisons locales modifiables. Les liaisons globales de TRAC suffisent, mais entraînent une perte de place-mémoire et une perte de souplesse.

Notons aussi que les modifications se font toujours explicitement et que l'environnement local détaillé au tableau 2.B est celui de chaque niveau d'expansion.

1.1.C.3 Accès aux liaisons

L'accessibilité des liaisons (en vue de délivrer la valeur ou de la modifier) dépend de leur portée :

1.- Liaisons globales : une fois créée et tant qu'elle n'est pas explicitement supprimée, une liaison globale est toujours accessible.

2.- Liaisons locales : les liaisons locales d'un niveau sont accessibles uniquement à ce niveau. Elles restent dans l'environnement pendant l'expansion des macros imbriquées, mais ne peuvent pas être accédées pendant ces temps. Le seul moyen d'accéder au niveau i+1 à une valeur créée au niveau i est donc de la passer comme argument.

L'exemple de la page suivante illustre brièvement les quelques points mis en évidence dans ce paragraphe. Le choix des conventions est une combinaison parmi les nombreuses possibles .

Les conventions suivantes étant adoptées :

- création et suppression implicites des arguments

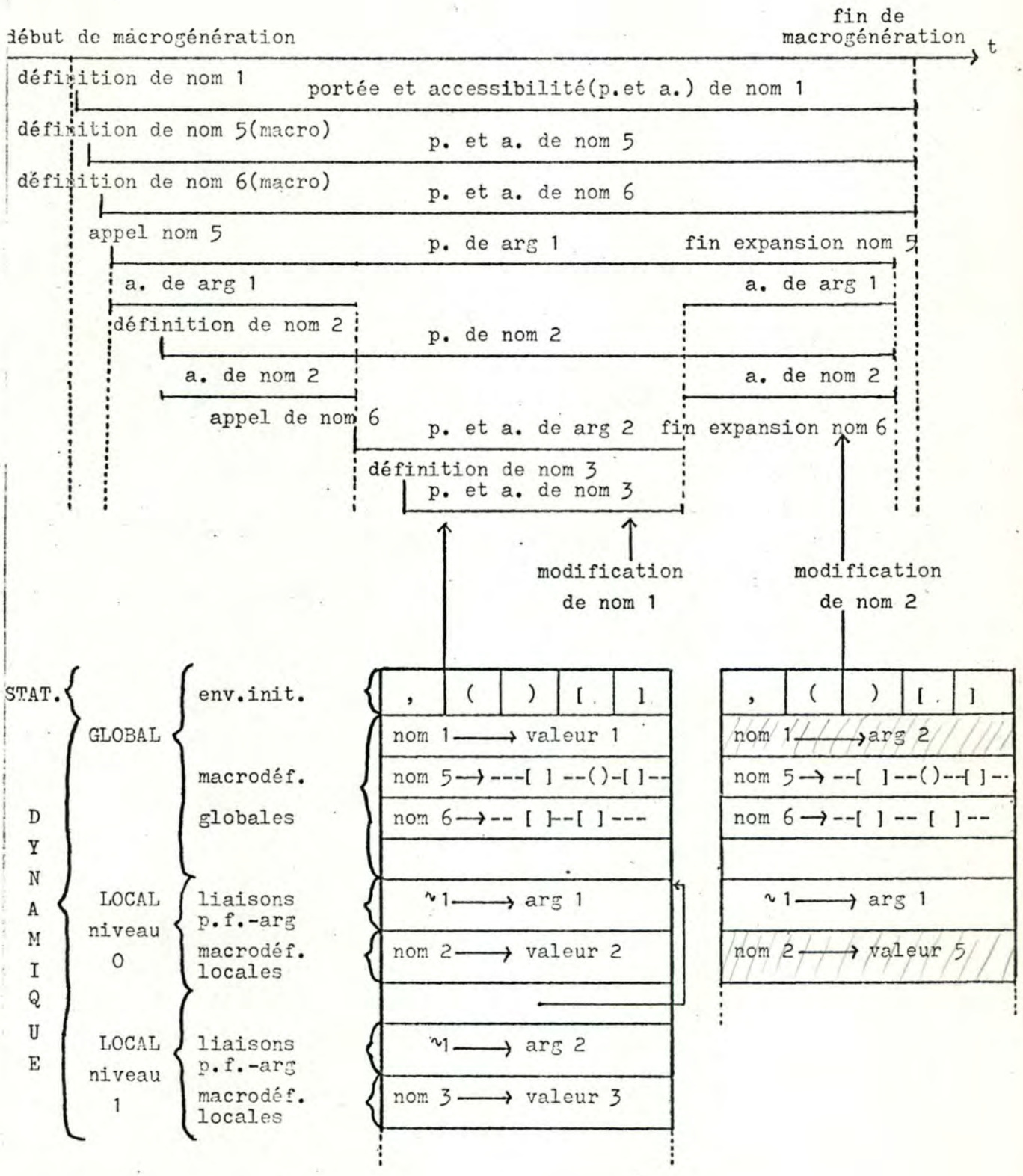
- création explicite et suppression implicite des macrodéfinitions (redéfinissables)

- portée des macrodéfinitions = celle du niveau où elles sont évaluées.

L'évaluation du texte

[nom 1, valeur 1] [nom 5, --[nom 2, valeur 2] -- (nom 6, arg 2) --[nom 3, valeur 5]--] [nom 6, --[nom 3, valeur 3] -- [nom 1, ~ 1] --] (nom 5, arg 1) --

suit le diagramme de temps :


début de macrogénération
fin de macrogénération
t
définition de nom 1
portée et accessibilité(p.et a.) de nom 1
définition de nom 5(macro)
p. et a. de nom 5
définition de nom 6(macro)
p. et a. de nom 6
appel nom 5
p. de arg 1
fin expansion nom 5
a. de arg 1
a. de arg 1
définition de nom 2
p. de nom 2
a. de nom 2
a. de nom 2
appel de nom 6
p. et a. de arg 2
fin expansion nom 6
définition de nom 3
p. et a. de nom 3
modification de nom 1
modification de nom 2
STAT.
DYNAMIQUE
GLOBAL
LOCAL niveau 0
LOCAL niveau 1
env.init.
macrodéf. globales
liaisons p.f.-arg
macrodéf. locales
liaisons p.f.-arg
macrodéf. locales
, ( ) [ ]
nom 1 → valeur 1
nom 5 → ---[ ] --()-[ ]--
nom 6 → -- [ ]--[ ] ---
~1 → arg 1
nom 2 → valeur 2
~1 → arg 2
nom 3 → valeur 3
nom 1 → arg 2
nom 5 → --[ ]--()--[ ]--
nom 6 → --[ ] -- [ ] --
~1 → arg 1
nom 2 → valeur 5

	STATIQUE (GLOBAL)	DYNAMIQUE GLOBAL						
		LIAISONS	MODIFIABLES	CREATION		SUPPRESSION		
				EXPLICITE	IMPLICITE	EXPLICITE	IMPLICITE	
ASS	environnement initial macrodéfinitions	variables globales	oui	1ère déclaration évaluée	-	-		
XPOP	environnement initial macrodéfinitions	conventions de notation	oui	pseudo-instructions(extérieures aux macrodéfin.)	-	pseudo-instructions avec opérande blanc (extérieures aux macrodéfin.)		FIN
GPM	environnement initial	macrodéfinitions + liste d'arguments	oui (update)	définition	-	-		
TRAC	environnement initial	macrodéfinitions	oui (redéfinissables)	définition	-	suppression(DD: delete definition)		DE
LIMP	les marqueurs définis en 1ère ligne (flagline) du texte source	macrodéfinitions	oui	définition	-	mise à blanc du Template Tree		MACROGENERATION
		variables	oui	définition	-	mise à blanc du Symbol Tree		
ML/I	environnement initial	constructions globales	non	définition	-	-		
		variables de macro génération	oui	-	début de macrogénération	-		
STAGE2	les marqueurs définis dans la flagline : ' indicateur de paramètre . fin de ligne source $ fin de ligne cible e escape macrodéfinitions	variables	oui	définition	-	-		

TABLEAU 2.A : ENVIRONNEMENT GLOBAL

	STATIQUE (LOCAL)	DYNAMIQUE LOCAL (chaque niveau)						
		LIAISONS	MODIFIABLES	CREATION		SUPPRESSION		
				EXPLICITE	IMPLICITE	EXPLICITE	IMPLICITE	
ASS	arguments	variables locales	oui	déclarations (début de macro-expansion)	-	-		FIN DE MACROEXPANSION
XPOP	arguments	conventions de notation	oui	pseudo-instructions(intérieures à macrodéfinition)	-	pseudo-instructions blanches (intérieures à macrodéfinition)		
GPM	arguments	macrodéfinitions ∈ liste d' arguments	oui	définition	-	-		
TRAC	arguments	-	-	-	-	-		
LIMP	-	arguments	oui	-	début de macro-expansion	-		
		arguments créés	oui	définition	-	-		
ML/I	arguments et délimiteurs	constructions locales	non	définition	-	macros-système de suppression		
		variables temporaires	oui		début de macro-expansion			
				-		-		
STAGE2	-	arguments	oui	-	début de macro-expansion	-		
		arguments créés	oui	définition	-	-		

TABLEAU 2.B : ENVIRONNEMENT LOCAL

1.2 Syntaxe des macrolangages

La syntaxe des macro-instructions est primordiale. Voyons pourquoi :

1.- une macro est définie une seule fois, mais peut être appelée des dizaines, voire des centaines de fois (cas des macros-système). Vu la fréquence des macro-instructions, il est essentiel que leur reconnaissance se fasse de manière efficace.

2.- nous pouvons toujours diviser le texte source en deux parties distinctes : le texte de définition (ensemble des macrodéfinitions) et le texte de génération (le reste). La liberté d'écriture du texte de génération dépend uniquement de la syntaxe des macro-instructions : suivant que celle-ci est rigide (recours à des caractères de contrôle) ou non, la notation du texte de génération sera stricte ou souple. Soulignons qu'en plus de la facilité d'écriture, l'indépendance de notation a comme avantage d'élargir le domaine d'application des macrogénérateurs (édition de texte, traduction, programmation mobile, ...) Nous trouvons ici ce qui a motivé la conception des macrogénérateurs les plus riches : STAGE2 et ML/I.

Lors de la conception d'un macrogénérateur, le choix de la méthode de reconnaissance des macro-instructions, et donc de leur syntaxe, est décisif. La syntaxe de la ligne prototype des macrodéfinitions et l'organisation (hash table, liste ou arbre) de la partie de l'environnement qui concerne les macrodéfinitions ~~se déduiront immédiatement de~~ SERONT CONDITIONNÉES PAR ce choix.

Ceci fait, il ne reste plus qu'à définir la syntaxe des textes de remplacement. Les choix sont ici très nombreux et ne portent pas à conséquence. Il suffira que la syntaxe exprime tous les mécanismes d'évaluation du macrogénérateur, et que ceci soit réalisé au moyen de conventions simples.

Nous utilisons les règles syntaxiques suivantes :

lettre ::= A/B/C/.../Z
chiffre ::= 0/1/.../9
spécial ::= ∿/./,/;/'/␣/x/*/+/-/$/e/(/)/</>/NL
caractère ::= lettre/chiffre/spécial
mot ::= lettre mot/chiffre mot/lettre/chiffre
atome ::= spécial/mot
string ::= / caractère string
corps ::= string

Le non-terminal nom désigne les éléments appartenant à un sous-ensemble de l'ensemble des mots : les restrictions (nombre maximum de caractères, caractère initial, ...) introduites sur la syntaxe des mots pour construire ce sous-ensemble ne sont d'aucun intérêt dans cette étude.

Les minuscules k, l, m et n désignent des entiers positifs.

Pour décrire la syntaxe des macro-instructions, nous recourons à la notation introduite par Brooker et Morris :

[constituant ?] désigne un constituant qui peut être omis;
[constituant *] désigne un constituant qui peut être répété;
[constituant * ?] désigne un constituant qui peut être omis ou répété.

{ constituant 1
  constituant 2
  -
  -
  -
  -
  -
  -
  constituant n } désigne les formes alternatives d'un constituant : une seule de ces formes peut être utilisée à la fois.

1.2.A Syntaxe des macro-instructions

La syntaxe des macro-instructions doit permettre au macrogénérateur :

1.- de les délimiter, c.à.d. de déterminer le début et la fin de chacune

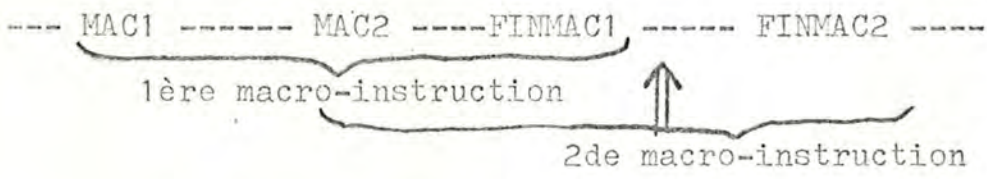
2.- d'identifier la macrodéfinition appelée

3.- de reconnaître les arguments

1.2.A.1 Délimitation des macro-instructions

Les conventions de délimitation des macro-instructions sont prédéfinies et font donc partie de l'environnement initial.

Un seul point est à souligner ici : aucun macrogénérateur n'a la possibilité de reconnaître deux ou plusieurs macro-instructions qui se recouvrent. Soit la situation suivante :



L'évaluation d'un tel texte se fait toujours comme suit : reconnaissance et expansion de la macro-instruction MAC1, suivi de l'évaluation du texte commençant en ⇑ .... ce qui entraîne éventuellement l'émission d'un message d'erreur et l'arrêt de la macrogénération !

1.2.A.2 Identification de la macrodéfinition appelée

Deux méthodes fondamentales existent :

1.- Identification par nom : chaque macro est identifiée par un nom. Celui-ci se trouve toujours à un endroit prédéterminé dans la macro-instruction (position fixe dans les macro-assembleurs, en début d'appel, ...). L'identification de la macro appelée est aisée : il suffit de comparer le nom apparaissant dans la macro-instruction avec les noms des macro-définitions qui se trouvent dans l'environnement.

2.- Identification par nom distribué(LIMP, STAGE2) : le nom de chaque macrodéfinition est subdivisé en substrings. Dans un appel, ces substrings sont séparés par les arguments. L'identification de la macrodéfinition appelée se fait donc conjointement avec la détermination des arguments, alors qu'ils se font séparément lorsque les macrodéfinitions sont identifiées par un nom.
Nous examinons cette méthode en 1.2.A.3.3

1.2.A.3 Reconnaissance des arguments

Il est essentiel que les arguments d'un appel soient reconnus et distingués les uns des autres. A cet effet, ils sont séparés par des délimiteurs et identifiés par leur position dans la macro-instruction.

Lorsque ces délimiteurs sont constitués par un marqueur prédéfini le mécanisme de reconnaissance des arguments est trivial : c'est le cas de GPM, où la syntaxe d'une macro-instruction est

$ nom [, string ¤] ;

Examinons les autres possibilités rencontrées.

1.2.A.3.1 XPOP ; notation étendue : le format des macro-instructions en "notation de base" est analogue à celui du MACRO-ASSEMBLER. Le recours à deux pseudo-instructions (CHPUNC définissant la ponctuation, NOISE définissant un ensemble de "noise words", c'est-à-dire de mots à ignorer) permet de rendre la notation du texte de génération très libre ("notation étendue") Ainsi, une macro-instruction qui aurait en notation de base le format

```
STORE       XX,YY
```

peut entre autres s'écrire, en notation étendue,

```
STORE YY INTO XX .
STORE INTO CELL 'XX' THE CONTENTS OF 'YY'.
```

etc...

Cette facilité de notation n'ajoute rien de fondamental. Du point de vue sémantique, l'effet est exactement le même que si les appels avaient été écrits en notation de base. La notation étendue n'est qu'un artifice permettant une écriture plus libre, mais non-signifiante.

Il n'en va plus de même dans les deux cas étudiés ci-dessous, où la syntaxe des macro-instructions détermine leur sémantique.

1.2.A.3.2 ML/I : structure de délimiteurs : la solution préconisée par P.J. Brown est sans aucun doute, parmi les méthodes d'identification par nom, la plus souple. L'utilisateur spécifie une structure de délimiteurs pour chaque construction qu'il définit. Comme le balayage du texte source se fait de gauche à droite sans retour en arrière, dès qu'un nom de construction est identifié, ML/I recherche les délimiteurs suivants jusqu'à ce qu'il trouve le dernier. Une construction étant ainsi reconnue, ML/I la traite, puis poursuit le scanning du texte source en cherchant le nom de construction suivant.

Un problème surgit ici : soit les noms de macro ADO, DO et DOP et le texte ADOPTE. Quelle est la macro appelée? Un mécanisme simple et rapide a été choisi pour éliminer de telles ambiguités : l'unité de base du texte source n'est pas le caractère, mais l'atome (cfr.règles syntaxiques en début de 1.2). Comme nous avons ici quatre atomes différents, aucune des trois macros n'est appelée.

La macro-système de définition de macro à la syntaxe suivante :

MCDEF [1 VARS ?] structure-de-délimiteurs AS corps;

Examinons la syntaxe de la structure-de-délimiteurs (1) (le reste ne nous concerne pas ici):

délimiteur ::= atome [{WITH / WITHS} atome *?]
nodeplace ::= N1
nodego ::= N1
delspec ::= [nodeplace ?] { délimiteur / OPT branche [OR[nodeplace ?] branche *?] ALL }
branche ::= délimiteur[delspec *?][nodego ?]
structure-de-délimiteurs ::= [delspec *] [nodego ?]

(1) Brown P.J. "ML/I User's Manual" pp. 5/1 à 12

Quelques remarques

- Une branche commence toujours par un délimiteur, le nom de branche

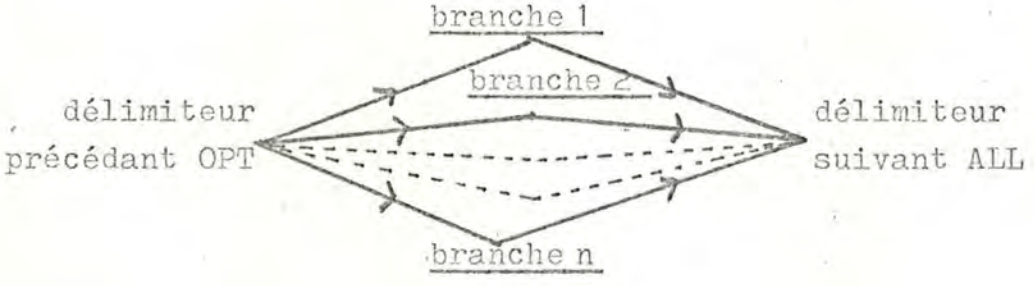
- Un délimiteur est constitué d'un atome ou d'une suite d'atomes : ceux-ci sont adjacents (option WITH) ou séparés par un nombre quelconque d'espaces (option WITHS)

- Les mots AS, OPT, OR, ALL, Nl, WITH et WITHS sont réservés : ils ne peuvent pas être utilisés comme délimiteurs.

L'examen de cette grammaire montre que la structure-de-délimiteurs est équivalente à un graphe orienté : les sommets sont constitués par les délimiteurs et les nodeplace (voir ci-dessous), les arcs indiquent les successeurs possibles de chaque délimiteur. Partant du sommet initial (délimiteur 0 ou nom de la construction), ML/I peut donc déterminer le successeur de chaque délimiteur jusqu'à ce qu'il trouve le dernier (terminateur de la construction).

Deux mécanismes fondamentaux sont à mettre en évidence : les listes à option et les noeuds.

Les listes à option permettent de spécifier les successeurs possibles d'un délimiteur (il va de soi qu'un seul de ces successeurs doit apparaître dans une macro-instruction). La forme d'une telle liste est OPT branche1 OR branche2 OR ..... OR branche n ALL ce qui se traduit par le graphe



Les noms des branches d'une même liste à option doivent être différents . Lorsqu'une branche ne se termine pas par un nodego, son successeur est le premier délimiteur qui succède à ALL dans le texte.

Les noeuds permettent de construire des circuits dans le graphe. Les restrictions suivantes (cfr. la grammaire) sont à respecter : un noeud (Nl) ne peut être placé (nodeplace) que devant un délimiteur ou une liste à option, et ne peut être atteint (nodego) qu'à partir de la fin d'une

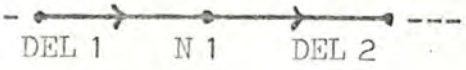
branche ou de la structure. Ces restrictions éliminent les ambiguités tout en laissant assez de souplesse.

Notons que les nodego et les nodeplace ne risquent jamais d'être confondus : un nodeplace précède toujours soit un délimiteur, soit OPT, alors qu'un nodego précède soit OR, soit ALL, soit AS. Nous pouvons assimiler un nodeplace à une étiquette et un nodego à un branchement inconditionnel ("go to"). On peut placer un nombre quelconque de noeuds dans une structure à condition que chaque nodeplace ait un indice l différent. Il n'y a pas de restrictions sur la portée d'un nodego : un tel "go to" permet de sauter à l'intérieur d'une liste à option quelconque ou d'en sortir sans aucune limitation autre que celle de "sauter" à l'intérieur de la structure.

Une étiquette "nodeplace" pouvant être atteinte à partir de n'importe quelle liste à option de la structure, il nous reste à examiner quels délimiteurs peuvent être atteints à partir d'un nodeplace. Nous distinguons trois cas :

1.- Le nodeplace précède un délimiteur extérieur à une liste à option :

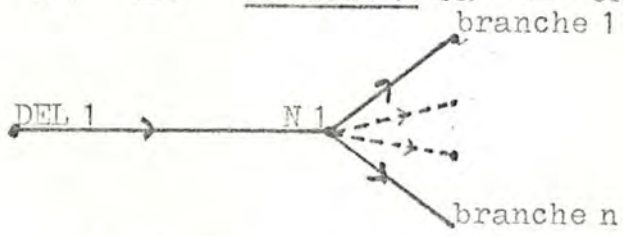
--- DEL 1 N 1 DEL 2 --- donne



2.- Le nodeplace précède une liste à option:

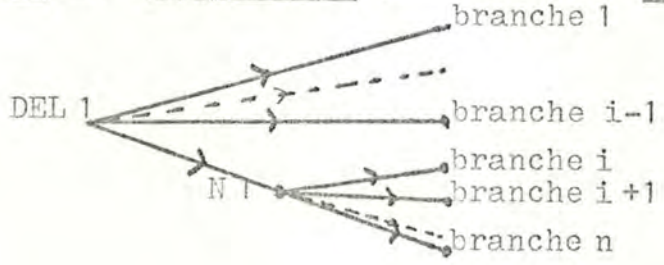
---- DEL 1 N 1 OPT branche 1 OR ---- OR branche n ALL

donne



3.- Le nodeplace précède un nom de branche dans une liste à option:

---- DEL 1 OPT branche 1 OR ---- OR branche i-1 OR
N 1 branche i OR branche i+1 OR ----- OR branche n ALL



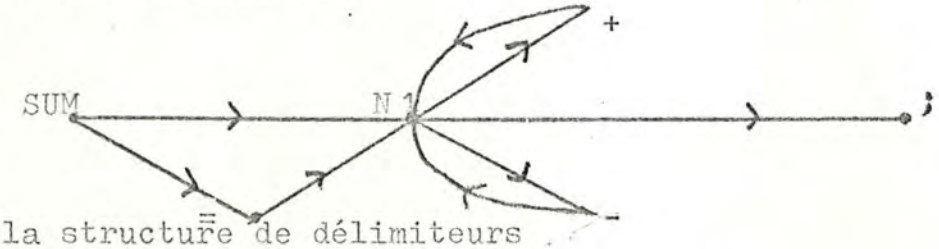
Dans le cas 2, toutes les branches de la liste à option peuvent être atteintes à partir du nodeplace N 1 ; dans le cas 3, seules les branches subséquentes au nodeplace N 1 dans le texte peuvent être atteintes à partir du N 1.

Deux exemples illustreront les possibilités offertes par cette méthode:

a.- soit une macro SUM permettant de calculer une somme de longueur quelconque et d'assigner optionnellement le résultat à une variable. Deux appels possibles à cette macro seraient entre autres :

```
SUM  A + B - C + D - E;

SUM  XX = Y + Z - OP;
```

le graphe orienté sera



et la structure de délimiteurs

SUM OPT = N 1 OR N 1 + N 1 OR - N 1 OR ; ALL

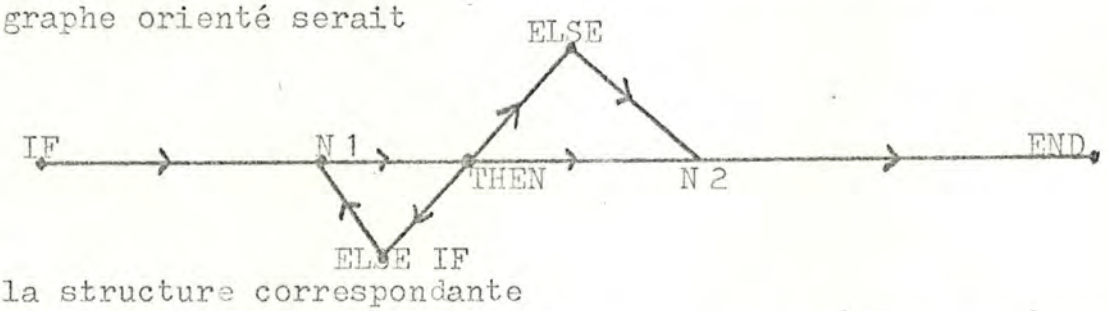
branche 1 branche 2 branche 3 branche 4

b.- soit une macro IF de format

IF ---- THEN ----{[ELSE --- ?]} END où

les arguments peuvent contenir une macro-instruction quelconque, excepté un IF imbriqué qui ne pourra apparaître qu'après ELSE.

Un graphe orienté serait



et la structure correspondante

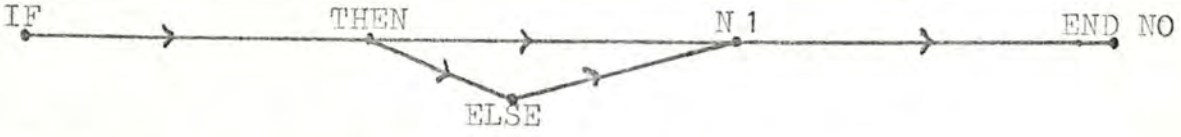
```
IF  N 1  THEN  OPT  ELSE  WITHS  IF  N 1  OR  OPT  ELSE  N 2
OR  N 2  END  ALL  ALL
```

Un appel à cette macro serait

IF condition 1 (arg 1) THEN SUM X = Y + Z; (arg 2) ELSE IF condition 2 (arg 3) THEN

SUM C = A - B ; (arg 4) END

Vu que les arguments peuvent contenir des macro-instructions et que des appels récursifs sont permis (cfr. paragraphe 1.3.B), nous pouvons obtenir le même effet avec une structure plus simple :



IF THEN OPT ELSE N 1 OR N 1 END NO ALL

Le noeud NO a une signification spéciale : il ne peut être utilisé que comme nodego après un délimiteur terminal, et indique que ce terminateur peut être commun à plusieurs macro-instructions imbriquées.

Le même appel que ci-dessous serait interprété comme suit :

IF condition 1 (arg 1) THEN SUM X = Y + Z; (arg 2) ELSE

IF condition 2 THEN SUM C = A + B; END

arg 3 = IF imbriqué (appelé récursivement si c'est spécifié dans le texte de remplacement).

END sert de terminateur aux deux IF imbriqués.

Nous constatons à propos des structures de délimiteurs que

1.- ce mécanisme permet un nombre variable d'arguments

2.- une seule macrodéfinition ML/I peut conduire à un nombre élevé de formes d'appel (les mécanismes d'évaluation des textes de remplacement doivent évidemment en tenir compte).

3.- l'écriture du texte de génération est souple. Une seule limitation : l'utilisateur peut spécifier uniquement la syntaxe de ce qui suit le nom de la macro, jamais celle du texte qui le précède.

1.2.A.3.3 LIMP et STAGE 2 : noms distribués : Cette méthode d'identification des macros et de détermination des arguments est basée sur un algorithme de rencontre de forme (pattern matching). Le texte source consiste en une suite de lignes : la ligne prototype ~~des~~ de chaque macro-définitions et les macro-instructions occupent une ligne (terminée par NL ou par le marqueur fin-de-ligne-source).

Etant entendu que

1.- chaîne est un string non vide

2.- chaîne ne peut pas contenir les marqueurs fin-de-ligne-source et indicateur-de-paramètre

la syntaxe de la ligne prototype suit la règle

[' ?] [chaîne ' * ?] [chaîne ?] [. ?] NL

Comme LIMP et STAGE 2 limitent la longueur du prototype à une ligne, une restriction conventionnelle a été ajoutée : le nombre d'indicateurs de paramètres est limité à 9. Etant donné les deux règles ci-dessus et celle qui suit

[constituant [n]] constituant pouvant être répété au plus n fois, la syntaxe de la ligne prototype se conforme donc à une des deux règles

[chaîne ' [9]] [chaîne ?] [. ?] NL (1)

' [chaîne ' [8]] [chaîne ?] [. ?] NL (2)

Quelques exemples montrent que le prototype est en fait un cliché (template)

IF' GT' DO' = 'NL (1a)

FOR' = 'STEP 'UNTIL 'DO 'END. (1b)

' = ' + ' . (2a)

' = MAX (' , ') NL (2b)

Une ligne du texte de génération est une macro-instruction si et seulement si elle rencontre un des clichés définis par l'utilisateur. Dans le cas contraire, elle est copiée dans le texte cible et le macrogénérateur passe à la ligne suivante. La rencontre est déterminée par l'algorithme de "pattern matching", qui contient donc les règles de rencontre.
Afin d'examiner ces règles, nous numérotons les clichés et convenons que chaque cliché D_i a une des formes suivantes :

$$\begin{cases} e_{i,0} \text{ ' } e_{i,1} \text{ ' } \cdots\cdots \text{ ' } e_{i,n_i} \text{ ' } [e_{i,(n_i+1)} \text{ ? }][\cdot ?]\ \underline{NL} \quad \text{(I)} \\ \text{ ' } e_{i,0} \text{ ' } e_{i,1} \text{ ' } \cdots\cdots \text{ ' } e_{i,n_i-1} \text{ ' } [e_{i,n_i} \text{ ? }][\cdot ?]\ \underline{NL} \quad \text{(II)} \end{cases}$$

avec $\begin{cases} 1 \leqslant n_i \leqslant 8 \\ e_{i,j} = \underline{\text{chaîne}} \qquad j=0, 1, \ldots, 18 \end{cases}$

Etant donné l'ensemble des clichés $D = \{D_i / \ i=1, 2, .., l\}$ une ligne S du texte de génération est une macro-instruction si, et seulement si S peut être décomposée en une suite $s_1 s_2 \cdots\cdots s_k$ de $\underline{\text{strings}}$ s_j et il existe un $D_i \in D$ tels que

$$\begin{cases} s_j = e_{i,j} & \forall j \text{ tel que } e_{i,j} \text{ apparaît dans } D_i \\ s_j = \underline{\text{string}} & \text{pour tout indicateur de paramètre de } D_i \end{cases}$$

Les strings s_k, numérotés dans leur ordre d'occurence, constituent les arguments de l'appel. Ceux-ci sont donc déterminés au cours du processus de rencontre.
Montrons que ce principe général de template matching ne suffit pas à supprimer toutes les ambiguités :

soit les clichés D_1 ' = ' = ' . (a)

D_2 ' = ' . (b)

et la ligne S A = B = C . (c)

D'après ce principe, la ligne (c) peut rencontrer ces clichés de trois manières :

	substring de S		substring de D_i	
(1) cliché (a)	s_1	A (argument 1)		'
	s_2	=	e_{10}	=
	s_3	B (argument 2)		'
	s_4	=	e_{11}	=
	s_5	C (argument 3)		'
(2) cliché (b)	s_1	A (argument 1)		'
	s_2	=	e_{20}	=
	s_3	B=C (argument 2)		'
(3) cliché (b)	s_1	A=B (argument 1)		'
	s_2	=	e_{20}	=
	s_3	C (argument 2)		'

Pour rendre le choix univoque, Waite (créateur de LIMP et STAGE 2) a ajouté un critère qui peut s'énoncer comme suit: le cliché D_i rencontré est celui dont chaque indicateur-de-paramètre correspond, dans l'ordre, à la plus courte sous-chaîne possible de S. Dans le cas présent, ce sera donc (a) : en effet, avec l'autre cliché, un des deux arguments aurait la configuration X=Y plus longue que celle, X, obtenue pour chacun des trois arguments avec (a). Si seul, le cliché (b) existait, nous aurions la situation (2), où le premier argument a rencontré la plus courte sous-chaîne plutôt que le second : l'ordre a donc été respecté.
Le choix de ce critère peut se justifier par une considération d'ordre intuitif (une enquête pourrait éventuellement la corroborer), suivant laquelle nos schèmes mentaux nous pousseraient, quand le choix nous en est laissé, à accorder entre eux le plus grand nombre d'éléments possible et aussitôt que possible.
Notons que les chaînes $e_{i,j}$ d'un cliché constituent bien un nom distribué.
L'organigramme de l'algorithme de "template matching" utilisé par STAGE 2 (cfr. l'annexe) donne une idée de la lenteur d'exécution du processus de template matching

1.2.A.4 Nombre variable d'arguments

Lorsqu'une macro-instruction ne peut pas avoir un nombre variable d'arguments, l'utilisateur est astreint à écrire autant de macro-définitions qu'il y a d'arguments possibles pour la macro-instruction en question ! Plusieurs macrogénérateurs possèdent donc un mécanisme permettant un nombre variable d'arguments : voyez le tableau 3.

Remarque

La correspondance paramètre formel - paramètre actuel est déterminée par la position des arguments dans la macro-instruction. Les paramètres-clé (key words) permettent de contrevenir cette convention : un argument n'est plus identifié par sa position, mais par un mot-clé qui le précède immédiatement dans le texte de la macro-instruction (XPOP, MACRO-ASSEMBLER).

1.2.B Syntaxe des macrodéfinitions

Dans le cadre de ML/I (structures de délimiteurs) et de STAGE 2 (template matching), la syntaxe jouait un rôle très important, puisqu'elle déterminait la sémantique des macro-instructions. En ce qui concerne les textes de remplacement, la distinction entre syntaxe et sémantique est beaucoup plus nette : les conventions syntaxiques sont préétablies. Leur énumération ne présente aucun intérêt. En particulier, les types des variables de macrogénération, les directives de calcul et de branchement conditionnel at macro-time seront examinées ultérieurement, dans le cadre de la sémantique des textes de remplacement.

Notons en passant que les paramètres formels dans les corps sont désignés soit par nom (ASS, XPOP, TRAC), soit par un nombre qui indique la position de l'argument correspondant dans l'appel.

	RECONNAISSANCE DES MACRO-INSTRUCTIONS (début/identification/fin)	RECONNAISSANCE et IDENTIFICATION des ARGUMENTS	NOMBRE D'ARGUMENTS
ASS	position : { nom dans zone code operation / arguments dans zone opérande	séparateur prédéfini ordre, sauf paramètre-clé	variable (listes et $SYSLIST)
XPO P	notation de base	position (comme ASS)	séparateur prédéfini ordre	fixe
	notation étendue	terminateur/1er atome non-NOISE qui soit un nom de macro/terminateur	atomes ≠ NOISE-words ordre, sauf emploi QWORD	fixe
GPM	marqueur-début/nom/marqueur-fin ← =argument 0	séparateur prédéfini ordre	fixe
TRAC	marqueur-début/nom/marqueur-fin	séparateur prédéfini ordre	variable grâce aux facilités de manipulation de strings
LIMP	ligne rencontrant un cliché (template matching)	par rencontre de cliché ordre	
M/L/I	warning mode	warning marker/nom/délimiteur terminal ← défini avec macro-système McWARN	structure de délimiteurs	variable (noeuds dans structure de délimiteurs)
/	free mode	← = délimiteur 0 nom/délimiteur terminal	ordre	
STAGE 2	ligne rencontrant un cliché	par rencontre de cliché ordre	variable (conversion de paramètre ed7 d' itération sur un string)

TABLEAU 3 : SYNTAXE DES MACRO-INSTRUCTIONS

1.3 Mécanismes d'évaluation

Nous pouvons résumer l'action globale du macrogénérateur en disant qu'il évalue le texte source. Avant d'entamer cette évaluation, il dispose de l'environnement initial, qui contient les différentes conventions syntaxiques ainsi que les actions que la détection de chaque convention doit entraîner. Au fur et à mesure de l'évaluation du texte source, l'environnement est modifié et des parties du texte source sont générées. Examinons les différents mécanismes qui sont à la base de l'évaluation.

1.3.A Conversions de paramètres

La présence d'arguments dans une macro-instruction serait superflue s'ils ne pouvaient être insérés dans les textes de remplacement.

Une "conversion de paramètre" indique qu'une information spécifique associée à l'argument référencé doit lui être substitué lors de l'évaluation du texte de remplacement où elle apparaît. Ces indications ne peuvent donc être écrites que dans les corps des macrodéfinitions.

Nous distinguons deux cas (cfr. conventions 1.1.B):

1.- Pendant la macro-expansion, les liaisons de l'appel étendu
~i ⟶ arg i sont dans l'environnement local.
L'occurence du paramètre formel " ~i" dans le texte de remplacement est remplacée par "arg i" lors de son évaluation. Ce mécanisme élémentaire existe dans tous les macrolangages.

2.- L'environnement contient en plus des liaisons
arg i ⟶ valeur i j
L'occurence du paramètre formel " ~i" accompagné d'un marqueur spécifique j indique que "valeur i j" doit lui être substitué. Cette seconde classe de conversions de paramètres n'existe qu'en MACRO-ASSEMBLER ("fonctions intrinsèques"), ML/I ("insertions") LIMP et STAGE 2 ("conversions").

Deux exemples illustrent quelques-unes des possibilités offertes:

a.- La fonction intrinsèque K' de MACRO-ASSEMBLER indique que la représentation décimale du nombre de caractères de l'argument référencé doit être généré :

```
macrodéfinition.  MACRO
                  NOMBRE    $A
                  ST        1, $A
                  ST        2, K' $A
                  MEND

l'appel  NOMBRE  ABC
génère           ST        1, ABC
                 ST        2, 3
```

b.- La syntaxe des conversions de paramètre en STAGE 2 est

conversion ::= e d i où

- e : est le marqueur escape
- $1 \leqslant d \leqslant 9$: d désigne la position de l'argument dans l'appel
- $0 \leqslant i \leqslant 8$: i indique le type d'insertion

ainsi i = 0 indique l'insertion de l'argument d

i = 4 indique que l'argument d doit être évalué comme une expression arithmétique et que le résultat doit être généré en format décimal.

```
macrodéfinition    COPIE*'*' .
                   e10
                   e24
                   $

l'appel  COPIE*ABC*6X(8-3)

génère             ABC
                   30
```

Remarque

Lorsqu'une conversion référence un argument volontairement omis, la "valeur par défaut" qui lui est substituée est généralement la chaîne vide. MACRO-ASSEMBLER et XPOP permettent la déclaration de valeurs par défaut dans la ligne prototype (cfr. tableau 4.A).

1.3.B Macro-instructions

Lorsqu'une liaison symbole ⟶ valeur est statique, la valeur est indépendante du moment où le symbole est référencé. C'est le cas des liaisons paramètre formel ⟶ paramètre actuel, lorsque les arguments ne contiennent pas eux-mêmes des macro-instructions.

Par contre, quand une telle liaison est dynamique, la valeur dépend du moment où la liaison est évaluée. Comme les modifications des liaisons et les références aux liaisons sont toujours explicites, cela ne pose normalement pas de problème.

En ce qui concerne les liaisons "paramètre formel ⟶ argument", cela en pose cependant un. Le mécanisme de passation des arguments étant implicite, il faut définir de manière univoque à quel moment (avant ou pendant la macro-expansion) les macro-instructions imbriquées dans les arguments sont étendues. P.J. Brown(1) cite trois méthodes courantes :

1.- Evaluation à priori (paramètres appelés par valeur immédiate) : un appel imbriqué dans un argument est évalué immédiatement et remplacé par sa valeur.

2.- Evaluation différée (appel par valeur différée) : tous les arguments de la macro-instruction sont délimités sans être évalués, et lorsque ceci est fait, immédiatement avant de commencer la macro-expansion, ils sont évalués et remplacés par leur valeur.

3.- Evaluation à posteriori (appel par nom) : les arguments sont délimités et passés textuellement , sans être évalués.

Pour illustrer la différence entre les trois méthodes, Brown donne l'exemple suivant en utilisant la notation GPM :

soit l'appel à deux arguments (le premier argument est une macro-instruction)

```
$ EXTER , $ INTER; , C;                    (1)
```

où le corps de la macrodéfinition INTER est "A, B"
les arguments passés lors de l'évaluation de (1) sont

---

(1) P.J. Brown : "A Survey of Macroprocessors" Annual Review in Automatic Programming, Vol 6, 1971, p. 61

méthode de passation	premier argument	second argument	troisième argument
par valeur immédiate	A	B	C
par valeur différée	A,B	C	(inexistant)
par nom	$ INTER	C	(inexistant)

Comme les passations par valeur imposent que l'évaluation des arguments se fasse très tôt, un mécanisme permettant d'inhiber l'évaluation est nécessaire (cfr. 1.3.F). L'avantage des passations par valeur est que les arguments sont évalués une fois pour toutes. D'autre part, elles permettent de générer dynamiquement des noms de macro : ainsi dans le texte (GPM)

$ DEF , NOM, ADD; $ DEF, ADD, ----; $$ NOM; , arg 1,....;

l'appel $$ NOM;, arg 1, ...; est équivalent à l'appel

$ ADD,arg 1 ,....;

L'appel par nom donne beaucoup plus de possibilités : il permet l'examen du"texte" des arguments aussi bien que celui des valeurs qui leur sont associées, et il permet aussi de modifier ces valeurs, tout cela en cours de macro-expansion.

A noter que les méthodes de passation sont prédéfinies dans chaque macrogénérateur.

Il nous reste à examiner de plus près l'évaluation des macro-instructions imbriquées dans les textes de remplacement. Dans tous les macrogénérateurs étudiés, ces macros sont évaluées dynamiquement, ce qui implique un stack pour l'environnement local et un stack pour la macro-expansion (les deux stacks pouvant éventuellement être réunis).

Remarquons qu'une macro-instruction imbriquée dans un corps n'a d'utilité que si elle peut être construite en se référant au contenu de l'environnement instantané, autrement dit si elle est évaluée en tenant compte du contexte. Sans cela, l'expansion dynamique serait un luxe dont la seule conséquence serait d'augmenter le coût de la macrogénération!

Le texte d'une macro imbriquée contient donc généralement des conversions de paramètres, qui doivent être évaluées au cours du processus de reconnaissance de la macro-instruction , mais antérieurement à la passation de ses arguments. La réalisation de ceci dépend de la manière dont le macro-générateur reconnaît les appels :

a.- la reconnaissance des macro-instructions se fait pas à pas, de gauche à droite, sans retour en arrière (GPM, ML/I). Le processus de reconnaissance consiste principalement à déterminerles délimiteurs successifs. Seuls les arguments peuvent être insérés : en effet, pour qu'un délimiteur inséré puisse être reconnu comme délimiteur, il faudrait le rescanner une fois son insertion réalisée.

b.- la macro-instruction imbriquée dans le corps est d'abord construite, et après cela elle est rescannée complètement (TRAC, LIMP, STAGE 2). Dans ce cas, le nom, les arguments et les délimiteurs peuvent être insérés en cours de construction. La possibilité du rescan est sous le contrôle de l'utilisateur : s'il inhibe le processus de rescan, le texte construit est considéré comme texte cible, et l'évaluation du texte source se poursuit en séquence.

Deux conséquences résultent de l'expansion dynamique :

1.- un texte de remplacement peut contenir des appels récursifs, à la condition qu'un mécanisme permettant de tester la fin de récursion soit présent. La réalisation d'un tel mécanisme suppose la présence de liaisons symbole-valeur locales ainsi que celle de directives de branchement conditionnel.

2.- l'expansion d'un appel imbriqué est en fait un branchement inconditionnel. C'est le seul mécanisme de branchement auquel recourent GPM et TRAC. Les autres macrolangages sont munis de directives de transfert permettant d'aiguiller le macrogénérateur vers un endroit déterminé du texte de remplacement en cours d'évaluation. Ce dernier type de transfert n'est pas indispensable : à condition d'être utilisé à bon escient, le branchement par appel imbriqué donne une puissance aussi grande.... son coût est cependant beaucoup plus élevé! Pour qu'une macro-instruction imbriquée soit étendue conditionnellement, la présence de variables et de directives de transfert conditionnel est évidemment requise.

1.3.C Macrodéfinitions

L'évaluation des macrodéfinitions est triviale. Elle se réduit à leur introduction (prototype + corps) dans l'environnement.

Imbriquer des macrodéfinitions dans un texte de remplacement n'a d'utilité que si elles sont introduites dynamiquement dans l'environnement. Deux avantages peuvent alors en résulter :

1.- de nouvelles macrodéfinitions peuvent être créées en cours de macrogénération, à la condition que le texte des macrodéfinitions imbriquées puisse contenir des conversions de paramètre et que celles-ci soient évaluées avant que la macrodéfinition ne soit introduite dans l'environnement. Sans cela, l'évaluation d'une macrodéfinition imbriquée se réduirait soit à ajouter une macrodéfinition déjà présente dans l'environnement, soit à en redéfinir une sans la modifier !

2.- les macrodéfinitions peuvent être utilisées comme liaisons symbole - valeur, la valeur étant de type string. Vu que la valeur des variables de type entier peut être mémorisée sous forme d'un string, ce seul type suffit pour permettre l'exécution de calculs at macro-time (à la condition que les directives de calcul fassent les conversions nécessaires !). Le gros inconvénient résultant de la limitation au type string est une perte d'efficience due à des temps de calcul plus longs.

1.3.D Variables de macrogénération

Rappelons que cette classe de liaisons permet de mémoriser des informations sur lesquelles des tests ou des calculs peuvent être exécutés. On peut s'en servir comme compteurs, comme indices pour des switchs ou des vecteurs, comme générateurs de constantes, etc.., Rappelons aussi que le dynamisme des macrodéfinitions et des macro-instructions fournit un dynamisme analogue, mais plus coûteux. En effet, les variables sont alors référencées par macro-instruction, ce qui entraîne une inefficience si le processus d'évaluation des macro-instructions est lent.

MACRO-ASSEMBLER est le seul macrolangage à posséder des déclarations explicites de variables de type booléen et de type entier. Vu l'utilité de ces derniers, ML/I est muni d'un mécanisme de création de variables entières. Dans tous les autres cas, le seul type disponible est le type string.

Parmi les variables de macrogénération, il en est une qui joue un rôle privilégié : le symbole généré. Il s'agit d'une variable globale permettant de générer des symboles propres à chaque macro-instruction étendue. En son absence, l'utilisateur peut toujours créer et gérer lui-même un tel symbole.

1.3.E Directives de macrogénération

Par directives de macrogénération , nous entendons les indications permettant d'effectuer des calculs et des transferts de contrôle at macro-time. Elles varient énormément d'un macrogénérateur à l'autre. Nous nous limitons donc à dégager les traits principaux.

1.3.E.1 Directives de calcul

Ils se répartissent en trois catégories :

a.- Arithmétique entière : tous les macrolangages offrent la possibilité d'effectuer des opérations arithmétiques élémentaires pendant la macrogénération. Certains se limitent à des directives de calcul sur deux opérandes (GPM, TRAC), d'autres permettent le calcul d'expressions arithmétiques (ASS, LIMP, ML/I, STAGE 2). Cela suffit pour le genre de calculs utiles at macro-time.

b.- Manipulation de strings : après les directives arithmétiques, celles de manipulation de chaînes de caractères sont sans doute les plus utiles. LIMP offre une grande souplesse dans ce domaine, par le fait que les textes de remplacement sont constitués d'instructions SNOBOL. Les macros-système TRAC de manipulation de pointeurs de balayage offrent elles aussi beaucoup de possibilités. D'autres macrolangages possèdent des directives d'extraction de substrings (ASS, ML/I, STAGE 2).

c.- Calcul booléen : des variables et directives booléennes sont rarement implémentées, vu que les directives d'arithmétique entière permettent d'obtenir des résultats analogues. MACRO-ASSEMBLER offre néanmoins de puissantes facilités à ce niveau, et TRAC possède des macros-système de calcul booléen sur un ou deux opérandes.

Remarque

XPOP tire profit de son langage de base, l'assembleur-FAP, en permettant

d'assembler et d'exécuter des séquences d'instructions-FAP at macro-time.

1.3.E.2 Directives de transfert

Les directives de transfert de contrôle offrent une grande souplesse pour indiquer des macrogénérations conditionnelles. Nous pouvons les subdiviser en trois catégories :

a.- Instruction de branchement du type

[IF condition ?] GO TO étiquette (ASS, LIMP, ML/I)

Les branchements ne peuvent être que locaux, c'est-à-dire intérieurs à chaque corps.

b.- Instructions conditionnelles du type (TRAC)

IF condition THEN $\left\{\begin{matrix}\text{constante}\\ \text{macro-instruction}\end{matrix}\right\}$ ELSE $\left\{\begin{matrix}\text{constante}\\ \text{macro-instruction}\end{matrix}\right\}$

A noter que GPM permet une facilité du même genre, basée uniquement sur le dynamisme des macrodéfinitions :

$ A, $ DEF, A, C ; , $ DEF, B, D ; ; équivaut à

IF A = B THEN D ELSE C

En effet, les macrodéfinitions étant mémorisées dans une liste et celle-ci étant examinée à partir de la fin, si les deux macrodéfinitions locales ont le même nom, c'est le corps de la dernière qui est évalué. Ce mécanisme à l'inconvénient d'être lent, donc coûteux, et assez recherché !

c.- Instructions de saut du type

[IF condition ?] SKIP n lignes $\left(\begin{matrix}\text{STAGE 2}\\ \text{XPOP, avec n=1}\end{matrix}\right)$

Dans le cas de STAGE 2, les lignes sont comptées suivant la séquence d'évaluation. Elles sont donc comptées à partir du niveau courant d'expansion i jusqu'à épuisement de n lignes, ce qui entraîne que l'évaluation se poursuit à un niveau j avec $0 \leqslant j \leqslant i$.

1.3.F Inhibitions des mécanismes d'évaluation

Dans certains contextes, il est souhaitable de pouvoir inhiber le processus d'évaluation. Lorsque l'utilisateur désire pouvoir recopier ou ignorer des chaînes de caractères, tels que des commentaires, leur évaluation risquerait de modifier l'évaluation du texte subséquent s'ils

contiennent des noms de macro. Lorsque nous voulons introduire une macro-définition dans l'environnement, nous désirons que le texte de remplacement y soit recopié littéralement : si les arguments (ligne prototype, corps) des macrodéfinitions sont évalués à priori, il faut pouvoir inhiber leur évaluation. Lorsque, suite à la réalisation de certaines conditions, une partie seulement d'un texte de remplacement doit être évalué , il faut pouvoir différer l'évaluation de ce texte jusqu'à la réalisation de ces conditions.

GPM et TRAC disposent dans ce but de marqueurs prédéfinis, les "parenthèses littérales". Le texte compris entre une parenthèse littérale ouvrante et la parenthèse fermante qui l'équilibre est recopié. L'évaluation d'un string entouré de parenthèses littérales équilibrées consiste donc simplement à supprimer celles-ci.

ML/I généralise ce mécanisme en laissant à l'utilisateur le soin de définir des marqueurs et des options d'inhibition par recours à la macro-système

MCSKIP [{M D T} * ?] structure-de-délimiteurs;

M définit un "matched skip" : toutes les inhibitions imbriquées sont reconnues et font partie du skip. En son absence, l'inhibition est un "straight skip" : le premier délimiteur final rencontré termine le skip.

D spécifie que les délimiteurs de l'inhibition doivent être copiés.

T spécifie que les arguments de l'inhibition doivent être copiés.

Soit le texte <A < B > C > D
Quelques définitions de skip seraient

1.- MCSKIP MT, <>; le skip imbriqué <B > est reconnu, l'argument A < B > C du skip est copié et l'évaluation se poursuit en D. Nous reconnaissons la définition des parenthèses littérales.

2.- MCSKIP T, <> ; l'argument A < B du "straight skip" est copié et l'évaluation se poursuit en C.

3.- MCSKIP MD, <>; les délimiteurs <> sont copiés, l'argument A < B> C est ignoré et l'évaluation se poursuit en D.

4.- MCSKIP M, <> ; l'évaluation se poursuit en D.

5.- MCSKIP <>; l'évaluation se poursuit en C.

Notons pour terminer que le contrôle du rescan (cfr. 1.3.B, paragraphe b) constitue aussi un procédé d'inhibition.

	CONVERSIONS DE PARAMETRES	PASSATION DES ARGUMENTS	VARIABLES DE MACROGENERATION
ASS	arguments fonctions intrinsèques(type, nombre, ...) V.P.D.(valeur par défaut) : {paramètres positionnels:vide {paramètres-clé:valeur déclarée dans le prototype	(quelconque)	variables globales et locales de type string, entier ou booléen créées et initialisées par des déclarations symbole généré:fonction intrinsèque $ SYSNDX
XPOP	arguments V.P.D.:paramètre formel	(quelconque)	-
GPM	arguments V.P.D.:vide	a priori	macrodéfinitions
TRAC	arguments V.P.D.:vide	a priori	macrodéfinitions
LIMP	conversions de paramètre(moins nombreuses qu'en STAGE 2) V.P.D.:vide	a posteriori	variables globales } type variables locales(=arguments créés) string symbole généré:conversion Oi
ML/I	arguments et délimiteurs valeurs résultant de l'évaluation des arguments et délimiteurs résultats d'expressions arithmétiques (variables permanentes et temporaires) V.P.D.:vide	macros-système:a priori macros-utilisateur:a posteriori	macrodéfinitions variables entières: {-permanentes:allouées et initialisées à 0 en début de macrogénération {-temporaires:3(ou plus) variables allouées et initialisées en début de macro-expansion, dont une peut servir comme symbole généré
STAGE 2	conversions de paramètre:e d i {1 ≤ d ≤9=numéro de l'argument {0 ≤ i ≤8=modificateur:indique le type de conversion(copie de l'argument,valeur de l'argument pris comme symbole,..) V.P.D.:vide	a posteriori	variables globales(créées par la fonction système e F3) type string variables locales (arguments créés par la conversion ed0) type string symbole généré:conversion e o i

TABLEAU 4.A : EVALUATION DU TEXTE

	DIRECTIVES DE CALCUL	DIRECTIVES DE TRANSFERT	INHIBITIONS
ASS	directives SET d'assignation {d'arithmétique entière(expressions) {de manipulation de strings {de calcul booléen(expressions)	directives de branchement {conditionnel(résultat d'expressions booléennes) {inconditionnel	-
XPOP	XECUTE = = COMPYL } code FAP assemblé et exécuté at macro-time	pseudo-instruction XPIFF permettant de sauter une ligne d'après condition	-
GPM	3 macros-système d'arithmétique entière(conversion binaire ↔décimal et opérations élémentaires sur deux opérandes)	un mécanisme basé sur les macro-définitions locales,équivalent à IF A=B THEN C ELSE D	parenthèses littérales
TRAC	macros-système {de manipulation de strings {d'arithmétique entière sur 2 opér. {de calcul booléen sur 1 ou 2 opér.	deux macros-système de comparaison équivalents à IF A {=, >} B THEN appel 1 ELSE appel 2	parenthèses littérales
LIMP	textes de remplacement = composés d'instructions SNOBOL	branchements {conditionnels, inconditionnels} de SNOBOL	inhibition du processus de template matching
ML/I	macro-système MCSET de calcul d'expressions arithmétiques macro-système MCSUB d'extraction de substring	macro-système MCGO de {branchement inconditionnel {branchement conditionnel(comparaison de strings et d'expressions arithmétiques)	définies par macro-système MCSKIP
STAGE 2	note:directive ::= eFi, appelée fonction-système itération sur un string(ed7 et eF8) itération par comptage(eF7 et eF8) arguments=expressions arithmétiques (ed4)	fonctions-système permettant de sauter n lignes {inconditionnellement(e F4) {conditionnellement(comparaison de 2 strings eF5,ou de 2 expressions arithmétiques eF6)	fonction-système d'inhibition de template matching (e F1) fonction-système de copie en output(eF2)

TABLEAU 4.B : EVALUATION DU TEXTE

1.4 Macrogénération différée

De ce que nous avons examiné jusqu'à présent, il résulte que les chaînes de caractères sont insérées dans le texte-cible à l'endroit où elles sont générées. Il serait intéressant de pouvoir assouplir la technique de macrogénération en permettant la séparation des phases de génération et d'insertion dans le texte-cible.

Tous les macrolangages étudiés ici sont assez puissants pour permettre l'écriture de macrodéfinitions réalisant un tel mécanisme (cfr. chapitre 2). L'incorporation de directives de macrogénération différée semble néanmoins la solution la plus efficace.

XPOP est le seul à fournir une pseudo-instruction de macrogénération différée, assez souple d'ailleurs :

{ WAITIN / WAIT } [étiquette ?] (1) / (2)

La partie de texte comprise entre la pseudo-instruction et la fin du texte de remplacement est évaluée et mémorisée

- (a) dans le cas où une étiquette est mentionnée
  - cas (1) : devant le texte déjà mémorisé dans la zone de mémoire réservée à cette étiquette
  - cas (2) : derrière le texte déjà mémorisé dans la zone de mémoire réservée à cette étiquette
- (b) dans le cas contraire
  - cas (1) : devant le texte déjà mémorisé dans une zone de mémoire réservée
  - cas (2) : derrière le texte déjà mémorisé dans une zone de mémoire réservée

Le texte mémorisé est inséré dans le texte-cible

- (a) à l'instant où l'étiquette est évaluée dans le texte de génération (niveau 0). Si elle est absente, le texte mémorisé est ignoré
- (b) en fin du texte-cible.

2. ILLUSTRATION DES TECHNIQUES DE MACROGENERATION PAR L'ECRITURE DE MACROS

2.1 Introduction

Après avoir dégagé les principes fondamentaux de la macro-génération, l'écriture de quelques macros permettra de mieux cerner les caractéristiques et les possibilités des macrolangages.

Avant d'aborder ces exemples, il est nécessaire d'expliciter certaines conventions propres à chaque macrolangage. Nous retenons uniquement les conventions qui nous seront utiles pour ces exemples.

Nous passons LIMP et XPOP sous silence (STAGE 2 est une extension de LIMP et XPOP ne se différencie pas fondamentalement d'un langage de macro-assemblage tel que celui d'IBM 360).

2.1.A MACRO-ASSEMBLER 360

&SYSLIST (i,j) param.formeli(j)	le jème sous-argument de l'argument i doit être substitué (cfr. ci-dessous et 2.3.A)
&SYSNDX	prend comme valeur, pendant l'expansion de la macrodéfinition où elle apparaît, la valeur décimale i si c'est le ième appel étendu depuis le début de la macrogénération
LCLA liste variables	déclaration de variables locales entières
GBLA liste variables	déclaration de variables globales entières
variable entière SETA expression arithmétique	le résultat de l'évaluation de l'expression est assigné à la variable
opérandes d'une expression arithmétique ou booléenne : paramètres formels, variables de macrogénération, constantes explicites (autodéfinies), fonctions intrinsèques	
AIF (expression booléenne).label	branchement conditionnel vers .label
AGO .label	branchement inconditionnel vers .label
MEXIT	branchement inconditionnel vers la fin du corps
ANOP	directive inffective (assembler no-operation)

Lors de l'expansion de MAC A,(B,C,D),E

l'évaluation de génère

&SYSLIST(1)	A
&SYSLIST(2,1)	B
&SYSLIST(2,2)	C
&SYSLIST(2,3)	D
&SYSLIST(3)	E

2.1.B GPM

&DEF,nom,corps;	macro-système de définition de macro
&UPDATE,nom,corps;	macro-système de redéfinition (la longueur du corps doit être ⩽ longueur du corps de la macrodéfinition référencée)
&VAL,nom,corps;	macro-système délivrant une copie littérale du corps de la macro référencée
∿ i	paramètre formel (une copie littérale de l'argument i doit lui être substitué)
< >	parenthèses littérales
&BIN,D; &DEC,B; &BAR,{+ - x /},B1,B2;	trois macros-système de calcul: conversion de la valeur décimale D en binaire conversion de la valeur binaire B en décimal addition, soustraction, multiplication, division : calcul binaire sur deux opérandes
&nom,arg1,...,argn;	macro-instruction

Directives de calcul décimal:

A partir des 3 macros-système de calcul, nous définissons 4 macros de calcul sur 2 opérandes décimaux :

```
&DEF,ADD,< &DEC,&BAR,+,&BIN,∿1;,&BIN,∿2;;;>;
&DEF,SUB,< &DEC,&BAR,-,&BIN,∿1;,&BIN,∿2;;;>;
&DEF,MUL,< &DEC,&BAR,x,&BIN,∿1;,&BIN,∿2;;;>;
&DEF,DIV,< &DEC,&BAR,/,&BIN,∿1;,&BIN,∿2;;;>;
```

L'appel &ADD,D1,D2; entraîne les actions suivantes :

1.- conversion de la valeur décimale D1 en binaire (BIN)

2.- conversion de la valeur décimale D2 en binaire (BIN)

3.- addition binaire des 2 opérandes convertis (BAR,+)

4.- conversion du résultat en décimal (DEC)

Rappelons que les arguments des macro-instructions (macros-système compris) sont passés par valeur immédiate. C'est la raison pour laquelle le second argument "&DEC,.....;;;" des 4 macrodéfinitions ci-dessus est enfermé dans des parenthèses littérales : sans cela,ce second argument (qui constitue le corps de la macro définie) serait évalué lors de la définition, ce qui n'aurait pas de sens et provoquerait des erreurs (dûes à l'absence d'arguments dans le cas présent: ∿1 et ∿2 ne pourraient être évalués).

Macrogénération conditionnelle:

&nom1,arg1,....,argn,&DEF,nom1,<corps 1>;,&DEF,nom2,<corps 2>;;

équivaut à

IF nom1 = nom2 THEN évaluer corps2 avec les arguments arg1,....,argn

ELSE évaluer corps1 avec les arguments arg1,....,argn

En effet, l'évaluation de la macro-instruction
"&nom1,arg1,.....< corps2 >;;" se fait comme suit :

- évaluation du nom"nom1", puis de chaque argument "arg1",...,"argn"
- introduction de la liaison "nom1 ⟶ corps1"dans l'environnement local
- introduction de la liaison "nom2 ⟶ corps2"dans l'environnement local
- évaluation de la macro "&nom1,arg1, ...,argn;"

si nom1 = nom2, c'est la dernière macro définie, c'est-à-dire nom2, qui est appelée.

Illustrons dès à présent ce mécanisme par un exemple :

soit une macrodéfinition de calcul de factorielle :

&DEF , FACT, < & ∿1 ,
&DEF , ∿1 ,< &MUL, ∿0 , &FACT , &SUB , ∿0 , 1 ;;; > ;, &DEF , 0 ,
< 1 >;; > ;

L'appel &FACT , 2 ; entraîne les actions suivantes :

- évaluation du nom " ∿1" ce qui donne "2"; pas d'arguments
- création de la liaison locale

  2 ⟶ &MUL , ∿0 , &FACT , &SUB , ∿0 , 1 ;;;
- création de la liaison locale

  0 ⟶ 1
- expansion de l'appel "&2" , ce qui donne
- &MUL , 2 , &FACT , 1 ;; (0 désigne le nom de l'appel courant)

donnera de la même manière

&MUL , 1 , &FACT , 0;;

l'expansion de cet appel créera deux liaisons locales de nom "0" , dont la seconde sera appelée ;
celle-ci génère 1 et termine la récursion.

&MUL , 1 , 1 ; donne 1

&MUL,2,1; donne 2, qui est bien la valeur de &FACT , 2 ;

Ce mécanisme, ainsi que les macros de calcul décimal seront souvent utilisés dans les exemples.

Remarque

Il est peut-être utile d'insister sur le fait qu'une macro-instruction n'est étendue qu'au moment où son délimiteur terminal ";" (qui balance son délimiteur initial "&") est rencontré.

2.1.C TRAC

&(DS,nom,string)	définition d'un string; le string est associé à son nom. De plus, un pointeur(le "form pointer") est associé au nom du string; sa valeur initiale pointe vers le début du string défini.Ce pointeur servira à permettre des manipulations sur les strings (cfr. ci-dessous)
&(SS,nom,par.formel 1,....., par.formel n;	segmentation du string nommé (cfr. ci-dessous)
&(nom,arg1,.....,argn) &&(nom,arg1,.....,argn)	macro-instruction mode actif macro-instruction mode neutre } ne modifient pas le "form pointer", sauf si nom = CR,CS,CC,CN ou IN
&(DD,nom 1,....,nom n)	suppression des définitions nommées
( )	parenthèses littérales (n'inhibent pas la substitution des arguments aux "gaps" < i>)
&(AD / SU / DV / ML ,D1, D2)	addition soustraction division multiplication } des deux valeurs décimales
&(EQ,X3,X4,X1,X2)	équivaut à IF string X3 = string X4 THEN X1 ELSE X2
&(GR,D1,D2,X1,X2)	équivaut à IF D1 >D2 THEN X1 ELSE X2
&(CR,nom)	positionne le f.p.(form pointer) en début du string nommé

MACROS-SYSTEME DE MANIPULATION DE STRINGS	RESULTAT		FORM POINTER APRES	
	NORMAL	ANORMAL	NORMAL	ANORMAL
&(CS,nom,Z)	segment suivant (de f.p. au gap <i> suivant)	si plus de segments,alors évaluer Z	1er caractère après le gap <i>	si plus de segments, fin du string
&(CC,nom,Z)	caractère sous f.p.(les < i > sont ignorés)	si plus de caractères, évaluer Z	caractère suivant	si plus de caractères, fin du string
&(CN,nom,D,Z)	substring de D caractères depuis f.p.(les <i> sont ignorés)	si moins de D caractères, alors évaluer Z	caractère suivant le dernier caractère du substring	si moins de D caractères,inchangé
&(IN,nom,X,Z)	substring depuis f.p. jusque substring identique à X	si pas de X dans la suite du string, alors évaluer Z	caractère suivant X dans le string	si pas de X, inchangé

Définition de macros:

&(DS,nom,string) & (SS,nom,par.formel 1, ... , par.formel n)

La première macro-système définit un string identifié par "nom".

La seconde segmente le string nommé d'après les arguments cités : les occurences du ième argument, c'est-à-dire de "par.formel i", dans le string sont remplacés par des marqueurs internes, lès "gaps" < i > . Lors de l'appel d'une telle macro, les arguments de l'appel sont substitués aux gaps <i> .

TRAC sépare donc la définition d'une macro et la définition de ses paramètres formels. Cette distinction permet d'ajouter ou de supprimer des paramètres formels du corps d'une macrodéfinition sans le moindre problème. Ceci est un trait tout à fait unique.

Exemple de macrodéfinition: &(DS,MAC,ABACABA) & (SS,MAC,B,C) crée la liaison MAC ⟶ A < 1 > A < 2 > A < 1 > A

L'appel &(MAC,X,Y) génère le string AXAYAXA

Remarque

La substitution des arguments aux < i > est opérée même lorsque l'évaluation des strings contenant des < i > est inhibée par des parenthèses littérales (ceci est spécifique à TRAC).

Macro-instructions:

- mode actif : &(nom,arg1,...., argn) le résultat de l'expansion, c'est-à-dire le string généré, est rescanné et évalué.
- mode neutre : &&(nom,arg1,...., argn) le résultat n'est pas rescanné.

La différence entre ces deux modes n'est réelle que pour les macro-instructions qui donnent lieu à un texte généré, c'est-à-dire pour

- certaines macro-instructions appelant des macros définies par l'utilisateur
- les macros-système
  - de manipulation de strings (CS,CN,CC,IN)
  - de calcul arithmétique (AD, SU,ML,DV)
  - de branchement par comparaison (EQ,GR)

par opposition à des macros-système DS,DD,SS,...

L'exemple suivant illustre l'action des parenthèses littérales et des 2 modes (rappel : les arguments sont toujours passés par valeur immédiate):

&(DS,AA,XXX) crée la liaison AA ⟶ XXX

&(DS,BB,(&(AA))) crée la liaison BB ⟶ &(AA)

Etant donné ces deux définitions,

&(DS,CC,(&(BB))) crée la liaison CC ⟶ &(BB)
l'évaluation du second argument supprime une couche de ( ).

&(DS,CC,&&(BB)) crée la liaison CC ⟶ &(AA)
l'évaluation du second argument génère &(AA) qui n'est pas rescanné

&(DS,CC,&(BB)) crée la liaison CC ⟶ XXX
l'évaluation du second argument délivre &(AA), qui est rescanné et génère XXX.

Manipulations de strings:

Les quatre macros-système CS, CC, CN, IN ont chacune deux effets :

- faire progresser le "form pointer"
- générer un substring (qui est rescanné ou non suivant le mode de la macro-système utilisée)

Macrogénération conditionnelle: deux macros-système

&(EQ,X3,X4,X1,X2) et &(GR,D1,D2,X1,X2)

dont les quatre arguments sont évalués AVANT la comparaison des deux premiers ... il y a donc intérêt à entourer X1 et X2 de parenthèses littérales. Le résultat de l'évaluation de X1 ou X2 est rescanné ou non suivant le mode de la macro-système de comparaison utilisée.

2.1.D ML/I

MCDEF [i VARS ?] structure-de-délimiteurs AS corps;	définition de macro-utilisateur (i=nombre de variables temporaires allouées lors de l'appel de cette macrodéfinition;à défaut, i=3) pour la structure-de-délimiteurs, cfr. 1.2.A.3.2
< >	parenthèses littérales définies par MCSKIP MT ,< > ; cfr. 1.3.F
∿.	délimiteurs d'insertion (cfr. ci-dessous)
Pi) avec Ti) i ::= constante/Ti/Pi MCSET {T, P} i = macro-expression;	variable permanente entière variable temporaire entière assignation du résultat de la macro-expression (cfr.ci-dessous) à la variable
∿Lj. (j ::= entier positif) ∿Lo. MCGO Lj; MCGO Lj {IF, UNLESS} arg1 {=, EN, GE, GR} arg2;	étiquette étiquette implicitement définie désignant la fin du corps branchement inconditionnel vers Lj. branchement conditionnel si (à moins que) {string arg1=string arg2 ; macro-expr arg1=m.e.arg2 ; macro-expr arg1≥m.e.arg2 ; macro-expr arg1>m.e.arg2
initialisation des variables temporaires en début de macro-expansion : T1 T2 T3 Ti i≠1,2,3 initialisation des variables permanentes Pi en début de macro-génération	 nombre d'arguments de l'appel étendu nombre d'appels rencontrés depuis le début de la macrogénération niveau d'imbrication de l'appel étendu 0 0

Conversions de paramètres:

La syntaxe des conversions de paramètres (appelées "insertions") est :

insertion ::= délimiteur texte délimiteur

texte ::= indicateur macro-expression / macro-instruction

indicateur ::= /WA/WD/A/D/L

macro-expression ::= primaire [{ + / - / x / / } primaire * ?]

primaire ::= [{ + / - } * ?] opérande

opérande ::= entier-non-signé / macro-variable

macro-variable ::= { P / T } indice

indice ::= entier-non-signé / macro-variable

L'évaluation d'une insertion se fait comme suit :

- si texte contient une macro-instruction, elle est étendue; l'évaluation de texte doit toujours délivrer un string de la forme
  indicateur macro-expression
- ensuite, la macro-expression est évaluée, ce qui donne un entier j
- ceci fait, l'indicateur signifie ce qui doit être inséré :

WA	insérer l'argument j de l'appel étendu
WD	insérer le délimiteur j de l'appel étendu (le délimiteur 0 est le nom de cet appel)
A	évaluer l'argument j et insérer le résultat
D	évaluer le délimiteur j et insérer son résultat
vide	insérer l'entier j

Les délimiteurs d'insertion sont définis par la macro-système MCINS.

Avant d'écrire les exemples, nous ajoutons donc les conventions suivantes à l'environnement initial :

```
MCINS   ~ . ;          délimiteurs d'insertion
MCSKIP MT, < >;        parenthèses littérales
```

2.1.E STAGE 2

marqueurs: '	indicateur de paramètre (cliché)
. ou NL	fin-de-ligne-source (clichés et texte de génération)
& ou NL	fin-de-ligne-cible (textes de remplacement)
e	escape
cliché cfr. 1.2.A.3.3	
corps	format d'une macrodéfinition
& fin de macrodéfinition	

$1 \leqslant d \leqslant 9$	CONVERSIONS DE PARAMETRE
e do	substituer l'argument d
e d1	substituer la valeur de l'argument d pris comme symbole; si ce symbole est absent de la table des symboles, générer le string vide
e d3	insérer le caractère qui suit l'argument d dans la ligne construite
e d4	insérer le résultat de l'argument d évalué comme une expression arithmétique
	LIAISONS VARIABLE ⟶ VALEUR
e oi	symbole généré (local à la macro étendue)
e F3	création d'une entrée dans la table des symboles (global) nom = argument 1 valeur= argument 2 (type string) ou modification de la valeur si l'entrée existait déjà
e d6	création ou modification d'un argument (local) nom = e do valeur= ligne construite (type string)
	ITERATIONS
e d7	initialise une répétition sur le string "ligne construite" (cfr. 2.3.E)
e F7	initialise une itération (la ligne construite est évaluée comme une expression arithmétique et donne le nombre d'itérations) (cfr. 2.5.E)
e F8	indique la fin de la portée d'une itération ou répétition

	TRANSFERTS DE CONTROLE
e F4	saut inconditionnel des n lignes suivantes, où n est le résultat de l'évaluation de l'argument 1 pris comme expression arithmétique (la ligne "&" de fin de macrodéfinition est ignorée dans le comptage; idem pour e F5 et e F6 ci-dessous)
e F5 {0 / 1}	soit n la valeur de l'argument 3 évalué comme expression arithmétique : si chaîne arg1 {= / ≠} chaîne arg2, sauter n lignes
e F6 {- / 0 / 1 / +}	si valeur de arg1 pris comme expr. arithmétique {< / = / ≠ / >} valeur de arg2 pris comme expr. arithmétique , sauter n lignes
e F9	saut vers la fin du texte de remplacement
e F1	en début de ligne : indique que la ligne suivante est un "cliché à format fixe" (cfr. 2.2.E) en fin de ligne : inhibe le processus de template matching; la ligne construite est sortie sans être rescannée (cfr. tous les exemples)

A partir de ces conventions , nous écrivons les macrodéfinitions suivan

```
' EQU '.          création ou mise à jour d'une variable globale
e F3
&

SKIP '.           saut inconditionnel de n lignes (n=résultat de
e F4              l'évaluation de arg1 pris comme expression arithmé-
&                 tique).
```

Dans toutes les macros qui suivent, arg3 est évalué comme une expressio arithmétique et donne la valeur n:

Saut conditionnel de n lignes sur comparaison des arguments 1 et 2 pris comme strings :

```
IF ' = ' SKIP ' .                                    égalité
e F50
&
IF ' \= ' SKIP '.                                    inégalité
e F51
&
```

Saut conditionnel de n lignes sur comparaison des arguments 1 et 2 évalués comme expressions arithmétiques :

```
                                                     arg 1          arg 2
IF ' LT ' SKIP ' .                                   <
e F6-
&
IF ' LE ' SKIP ' .                                   ≤
IF e 10 LT e 20 SKIP e 30+1
e F60
&
IF ' EQ ' SKIP ' .                                   =
e F60
&
IF ' NEQ ' SKIP ' .                                  ≠
e F61
&
IF ' GT ' SKIP ' .                                   >
e F6+
&
IF ' GE ' SKIP ' .                                   ≥
IF e 10 GT e 20 SKIP e 30+ 1
e F60
&
```

Certaines de ces macros seront utilisées dans plusieurs exemples.

2.2 Code à format fixe - Symbole généré

Il arrive assez fréquemment que l'on veuille générer du code à format fixe, par exemple une suite d'instructions dans un langage d'assemblage.

Soit à générer du code pour calculer la valeur absolue d'un nombre.

2.2.A ASS.

Macro-définition :

```
                MACRO
                ABS     &X,&A
                L       1,&A
                LTR     1,1
                BC      2,E&SYSNDX    branchement si positif
                LCR     1,1
E&SYSNDX        ST      1,&X
                MEND
```

L'appel ABS Z,M génère, si c'est le 7ème appel rencontré ,

```
                L       1,M
                LTR     1,1
                BC      2,E7
                LCR     1,1
E7              ST      1,Z
```

Le code généré est d'office à format fixe dans un macro-assembleur.

2.2.B GPM

Ici, comme en TRAC, nous devons gérer nous-mêmes le symbole généré. De plus, pour générer en format fixe, nous devons tenir compte

- du nombre de chiffres du symbole généré
- des caractères "espace" et "NL" (newline)

Ceci est vrai aussi pour TRAC et ML/I. Pour simplifier, nous limitons le nombre d'appels à 99 : le symbole généré comporte 1 ou 2 caractères.

Comme GPM ne possède pas de directives de comparaison, nous initialisons ce symbole à 9, de façon à ne pas devoir tester le nombre de

Note : pour le besoin de la cause, nous n'avons pas utilisé l'instruction-assembleur LPR dans l'exemple.

chiffres :

```
&DEF,SYGEN,09;
&DEF,ABS,<&UPDATE,SYGEN,&ADD,&SYGEN;,1;;
    L    1,~2
    LTR  1,1
    BC   2,E&SYGEN;
    LCR  1,1
E&SYGEN; ST ~1, 1>;
```

Note

Il eût été possible de considérer des symboles générés de longueur variable, mais au prix d'une complexification considérable de la macrodéfinition.

2.2.C TRAC

Le recours à la macro-système &(GR,....) utilisée en mode actif, permet de tester le nombre de caractères du symbole généré et de générer du code en fonction de ce nombre.

```
&(DS,SYGEN,0)
&(DS,ABS,(&(DS,SYGEN,&(AD,&(SYGEN),1))
    L    1,X2
    LTR  1,1
    BC   2,E&(SYGEN)
    LCR  1,1  &(GR,&(SYGEN),9,(
E&(SYGEN) ST   1,X1),(
E&(SYGEN)  ST   1,X1))))
&(SS,ABS,X1,X2)
```

L'appel &(ABS,X,A) en TRAC ou &ABS,X,A; en GPM génère, si SYGEN valait au moment de l'appel :

SYGEN=3 (exclu en GPM, puisque initialisé à 9)

```
    L    1,A
    LTR  1,1
    BC   2,E4
    LCR  1,1
E4  ST   1,X
```

SYGEN=27

```
    L    1,A
    LTR  1,1
    BC   2,E28
    LCR  1,1
E28 ST   1,X
```

2.2.D ML/I

Nous pouvons utiliser la valeur initiale de la variable temporaire T2 comme symbole généré. La macro-système { MCSKIP /WITH/ ; } définit un "skip" (cfr. 1.3.F) qui ignore tout ce qui est compris entre // et le début de la ligne suivante, le caractère NL inclus.

```
MCDEF ABS WITH ( , ) AS <
    L    1,∿WA2.
    LTR  1,1
    BC   2,E∿T2.
    LCR  1,1
MCGO L1 IF ∿T2. GR 9;>// if sygen>9 goto ∿L1.
<E∿T2.␣␣ST 1,∿WA1. MCGO L0; > //   sygen ≤9
<∿L1.E∿T2.␣ST 1,∿WA1. >;
```

Un appel peut s'écrire ABS(X,A)

2.2.E STAGE 2

L'occurence de e F1 en début de ligne indique que la ligne suivante est un "cliché à format fixe" : les séquences de chiffres identiques apparaissant dans cette ligne sont interprétées comme des zones où les arguments référencés doivent être copiés à partir de la gauche (avec troncature à droite).

```
'=ABS(').
    L    1,e20eF1
    LTR  1,1eF1
    BC   2,Ee00eF1
    LCR  1,1eF1
1e86              arg.8:=1
e00e96            arg.9:=valeur du symbole généré
eF1
E99 ST   8,111111
&
```

X=ABS(A) génère si le symbole généré(e00)vaut 3 au moment de l'appel

```
      L    1,A
      LTR  1,1
      BC   2,E3
      LCR  1,1
E3    ST   1,X
```

IDENTIFIER=ABS(VARIABLE) génère si symbole généré=134

```
      L    1,VARIABLE
      LTR  1,1
      BC   2,E134
      LCR  1,1
E13   ST   1,IDENTI
```

Remarque

La prise en considération des caractères "espace" et "NL" est assez secondaire. S'y attarder de trop près risquerait de masquer les problèmes que nous voulons mettre en évidence dans les exemples suivants. Nous n'en tiendrons donc plus compte dans ce qui suit.

2.3 Nombre variable d'arguments

Nous avons souligné l'intérêt d'un tel mécanisme en 1.2.A.4 . Soit donc à générer du code pour une macro-instruction de type

X:=A1±A2±.....±An

où le signe du 1er opérande, A1, est implicitement +.

2.3.A ASS.

La fonction intrinsèque N' délivre le nombre de "sous-arguments' d'un argument-liste. Ainsi si (A,B,C) est l'argument correspondant au paramètre formel &A, alors N'&A est égal à 3.

```
     MACRO
&RES CALC   &LISTE
     LCLA   &CT
     L      2,&LISTE(1)
&CT  SETA   1
.DEB ANOP
&CT  SETA   &CT+1
     AIF    (&CT GT N'&LISTE).FIN
     AIF    (&LISTE(&CT) EQ '-').MIN
&CT  SETA   &CT+1
     A      2,&LISTE(&CT)
     AGO    .DEB
.MIN ANOP
&CT  SETA   &CT+1
     S      2,&LISTE(&CT)
     AGO    .DEB
.FIN ST     2,&RES
     MEND
```

Note

En remplaçant &LISTE(1) par &SYSLIST(1,1) et &LISTE(&CT) par &SYSLIST(1,&CT), nous obtenons le même résultat.

```
X    CALC   (A,+,B,-,CDE)   génère
     L      2,A
     A      2,B
     S      2,CDE
     ST     2,X
```

2.3.B GPM

Ne disposant pas de cette facilité, ni de directives de manipulation de strings, nous devons écrire autant de macrodéfinitions que de cas possibles : un appel aura le format

&CALCj,RESULT,X2,±,X4,.....,±,Xj;

le nombre d'arguments d'un appel sera toujours pair et doit être indiqué dans le nom de la macro pour que la macrodéfinition correspondante soit sélectionnée.

Une solution possible :

```
&DEF,+,<ADD   ∿1>;
&DEF,-,<SUB   ∿1>;
&DEF,CALC2,<LOAD   ∿2
           STORE  ∿1>;
&DEF,CALC4,<LOAD   ∿2
           &∿3,∿4;
           STORE  ∿1>;
&DEF,CALC6,<LOAD   ∿2
           &∿3,∿4;
           &∿5,∿6;
           STORE  ∿1>;
etc...
```

L'appel &CALC6,X,A,+,B,-,C; entraîne les actions suivantes :

génération de	LOAD	A
appel de &+,B; qui génère	ADD	B
appel de &-,C; qui génère	SUB	C
génération de	STORE	X

2.3.C TRAC

Ce macro-langage étant doté de puissantes facilités de manipulation de strings, plusieurs solutions sont possibles. Nous en proposons deux. Pour chacune, un appel serait par exemple &(CALC,RESU,AB-CD+EF-GH)

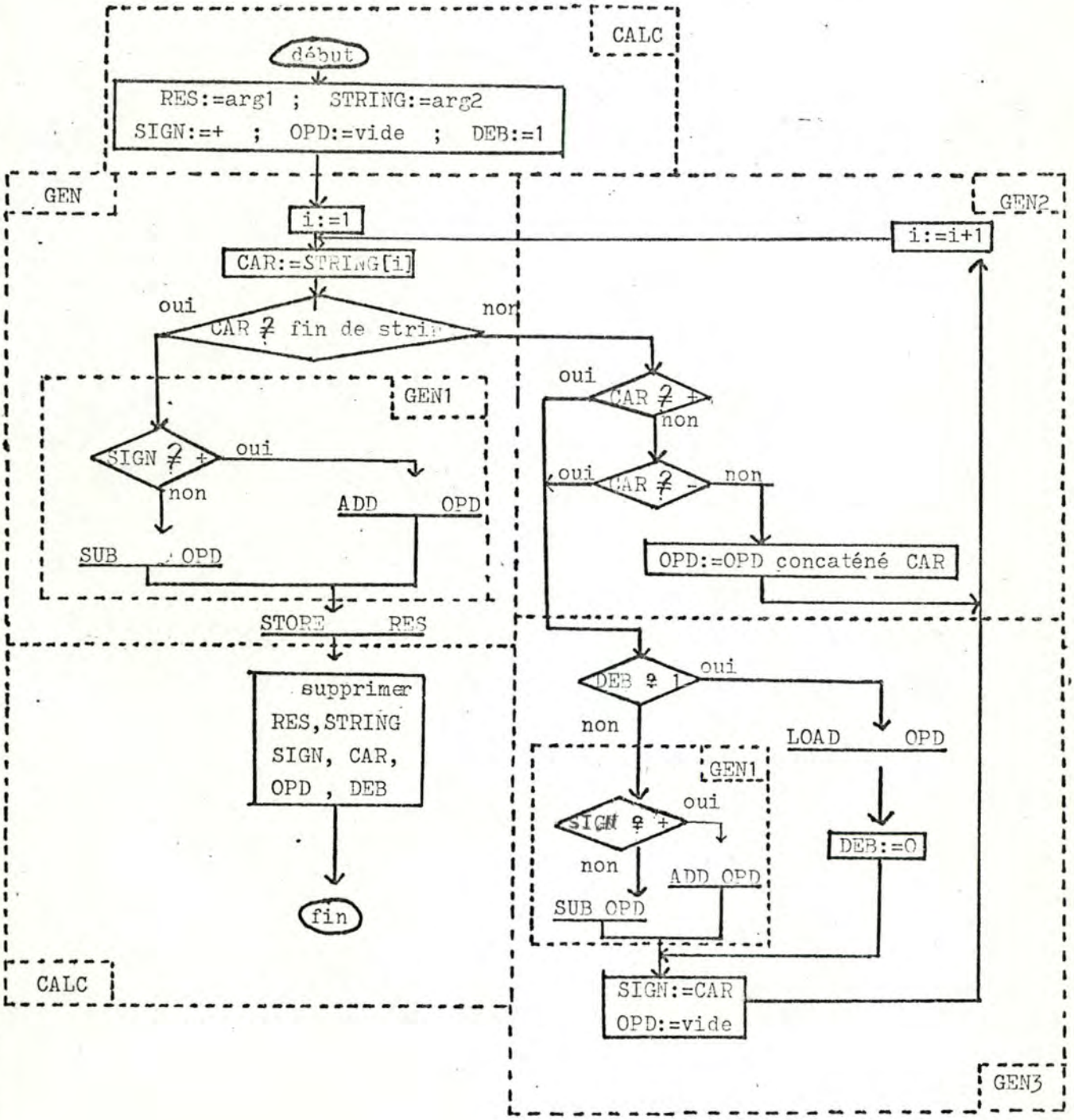
a.- analyse caractère par caractère:

Nous appellerons STRING le second argument formé par l'expression arithmétique donnée (AB-CD+EF-GH dans notre exemple). Ce string sera balayé de gauche à droite, caractère par caractère, par la macro GEN. Les différents opérandes seront déterminés par concaténation des caractères ainsi rencontrés jusqu'à une occurence de l'un des caractères '+' ou '-' (macro GEN2). Lorsqu'un de ces caractères est rencontré, la macro GEN3 va tester s'il s'agit du 1er opérande (au moyen de la variable DEB initialisée à 1) et générer soit un LOAD, soit faire appel à GEN1, puis mémoriser le signe rencontré dans la variable SIGN et réinitialiser la variable OPD (contenant l'opérande courant) à vide. La macro GEN1 générera, à l'aide des macros '+' et '-', les instructions d'addition et de soustraction du dernier opérande en fonction du signe qui le précédait (mémorisé par GEN3 dans la variable SIGN) et en tenant compte du nom de cet argument (mémorisé dans OPD par GEN2). La macro CALC a pour rôle d'initialiser les variables de travail STRING, SIGN, OPD et DEB, et de donner à la variable RES une valeur égale au nom de la zone résultat (premier argument de l'appel, RESU dans notre exemple).

A noter l'utilisation de la macro-système CC qui a pour effet de balayer le string STRING caractère par caractère, délivrant comme résultat soit le caractère suivant du string balayé, soit, si ce balayage est terminé, le caractère '*' qui servira à tester la fin de la génération.

Enfin, l'utilisation de la macro-système DD dans la macro CALC n'a pour but que de libérer les zones-mémoire utilisées pour les strings RES, OPD, etc...

Dans l'organigramme qui suit, i représente le "form pointer". Il est géré implicitement par les macros-système de manipulation de strings.


CALC
début
RES:=arg1 ; STRING:=arg2
SIGN:=+ ; OPD:=vide ; DEB:=1
GEN
i:=1
CAR:=STRING[i]
oui
CAR ≠ fin de strin
non
GEN1
SIGN ≠ +
oui
non
ADD OPD
SUB OPD
STORE RES
supprimer
RES, STRING
SIGN, CAR,
OPD , DEB
fin
CALC
GEN2
i:=i+1
oui
CAR ≠ +
non
oui
CAR ≠ -
non
OPD:=OPD concaténé CAR
DEB ≠ 1
oui
non
LOAD OPD
GEN1
SIGN ≠ +
oui
non
ADD OPD
SUB OPD
DEB:=0
SIGN:=CAR
OPD:=vide
GEN3

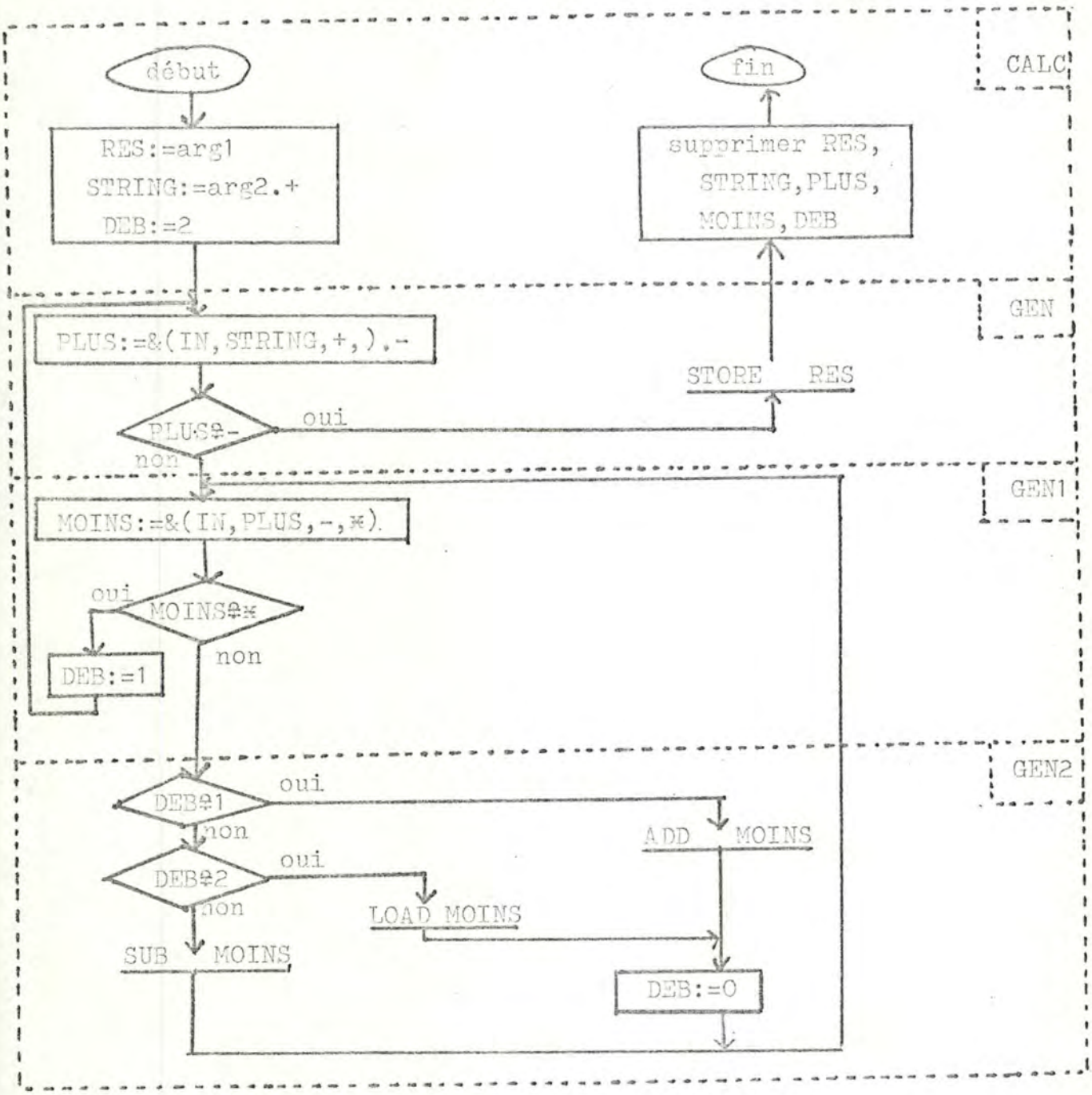
Nous obtenons :

```
&(DS,+,(ADD   X1))&(SS,+,X1)
&(DS,-,(SUB   X1))&(SS,-,X1)

&(DS,CALC,(&(DS,RES,X1)&(DS,STRING,X2)&(DS,SIGN,+)
          &(DS,OPD,)&(DS,DEB,1)&(GEN)
          &(DD,RES,STRING,SIGN,OPD,DEB,CAR)))
 &(SS,CALC,X1,X2)

&(DS,GEN,(&(DS,CAR,&&(CC,STRING,*))
         &(EQ,&(CAR),*,(&(GEN1)STORE   &(RES)),
                       (&(GEN2)))))

&(DS,GEN1,(&(EQ,&(SIGN),+,(&(+,&(OPD))),(&(-,&(OPD))))))

&(DS,GEN2,(&(EQ,&(CAR),+,(&(GEN3)),
                         (&(EQ,&(CAR),-,(&(GEN3)),
                                        (&(DS,OPD,&&(OPD)&&(CAR)))))))
          &(GEN)))

&(DS,GEN3,(&(EQ,&(DEB),1,(LOAD   &(OPD)&(DS,DEB,0)),
                         (&(GEN1)))
          &(DS,SIGN,&&(CAR))&(DS,OPD,)))
```

b.- Segmentation sur les signes + et -

Dans cette dernière forme, au lieu de balayer caractère par caractère, on emploie la macro-système IN pour aller rechercher directement le signe '+' ou '-' suivant. Comme cette macro-système ne permet de définir qu'un seul symbole de segmentation, il est nécessaire de rechercher d'abord le premier '+' suivant, puis de segmenter le sub-string ainsi obtenu sur le caractère '-'.

Afin d'éviter les problèmes en fin de balayage, l'expression donnée est concaténée à gauche d'un caractère '+' supplémentaire.On a procédé de même pour chaque substring en le concaténant à gauche d'un caractère '-'.Dans l'organigramme qui suit, l'opérateur de concaténation est représenté par un point(mais il est implicite dans le langage TRAC).



Dans les deux cas, l'appel &(CALC,RESULT,AB-CD+EF-GH)

génère

```
LOAD    AB
SUB     CD
ADD     EF
SUB     GH
STORE   RESULT
```

Nous obtenons :

```
&(DS,CALC,(&(DS,RES,X1)&(DS,STRING,X2+)&(DS,DEB,2)
          &(GEN)&(DD,RES,STRING,DEB,PLUS,MOINS)))
 &(SS,CALC,X1,X2)

&(DS,GEN,(&(DS,PLUS,&&(IN,STRING,+,)-)
         &(EQ,&(PLUS),-,(STORE   &(RES)),(&(GEN1)))))

&(DS,GEN1,(&(DS,MOINS,&&(IN,PLUS,-,*))
          &(EQ,&(MOINS),*,(&(DS,DEB,1)&(GEN)),(&(GEN2)))))

&(DS,GEN2,(&(EQ,&(DEB),1,(ADD   &(MOINS)&(DS,DEB,0)),
                         (&(EQ,&(DEB),2,(LOAD   &(MOINS)&(DS,DEB,0)),
                                        (SUB   &(MOINS)))))
          &(GEN1)))
```

2.3.D ML/I

```
MCDEF CALC :WITH= N1 OPT + N1 OR - N1
  OR ; ALL AS
<LOAD  ~WA2.
 MCSET T3=2;
 ~L4. MCGO L1 IF ~DT3.=+;
      MCGO L2 IF ~DT3.=-;
      MCGO L5;
 ~L1.  ADD   ~WAT3+1.  MCGO L3;
 ~L2.  SUB   ~WAT3+1.
 ~L3. MCSET T3=T3+1;   MCGO L4;
 ~L5.  STORE ~WA1. >;
   CALC  X := AB + C - D + EF ;
élimiteurs: 0      1    2   3   4    5
rguments:       1     2   3   4   5
```

génère

```
        LOAD   AB
        ADD    C
        SUB    D
        ADD    EF
        STORE  X
```

2.3.E STAGE 2

L'occurence de ed7 en fin de ligne initialise une répétition sur un string au moyen de caractères de coupure (break characters).

Le string est la ligne construite précédant l'occurence de ed7, et les caractères de coupure sont ceux qui suivent ed7 :

ligne construite ed7 caractères de coupure
} portée de chaque répétition
eF8

Au début de chaque répétition :

- un nouvel argument d est créé : le segment compris entre le début et le 1er caractère de coupure suivant de la ligne construite.
- ce segment est supprimé de la ligne construite.

Lorsque la ligne construite est vide, l'évaluation se poursuit avec la ligne qui suit eF8.

Il est utile de rappeler que les macros EQU, IF.... et SKIP ont été définies en 2.1.E.

```
':='                    ligne prototype                          (1)
OPe96                   argument9:=OP
OP EQU 0                OP:=0  (initialisation de OP)
e20e57+-                argument5:= segment  courant de argument2.
IF e91 = + SKIP 3       si OP vaut +
IF e91 = - SKIP 4       si OP vaut -
LOAD   e50eF1
SKIP 3
ADD    e50eF1
SKIP 1
SUB    e50eF1
OP EQU e53              mise à jour de OP;valeur de OP:=caractère qui suit
eF8                     l'arg.5 dans la ligne construite, c-à-d. l'opéra-
STORE  e10eF1           teur '+' ou '-' suivant.
&                                un appel serait par exemple X:=AB-CD
```

(1) Waite W.M. and Poole P.C. "The STAGE 2 Macroprocessor User's Manual" p. 42.

2.4 Récursivité

Soit à résoudre par macrogénération le problème des tours de Hanoï.

2.4.1 ASS.

&K = nombre de disques à transférer de la tour &A à la tour &B, les trois tours étant numérotées 1, 2 et 3.

```
     MACRO
     HANOI  &K,&A,&B
     LCLA   &X,&Y
     AIF    (&K LT 1).FIN
&X   SETA   &K-1
&Y   SETA   6-&A-&B
     HANOI  &X,&A,&Y
     DC     C'MOVE FROM &A TO &B'
     HANOI  &X,&Y,&B
.FIN MEND
```

L'appel HANOI 2,1,2

génère
```
     DC   C'MOVE FROM 1 TO 3'
     DC   C'MOVE FROM 1 TO 2'
     DC   C'MOVE FROM 3 TO 2'
```

2.4.2 GPM

```
&DEF,HANOI,<&~1,~2,~3,
  &DEF,~1,<&HANOI,&SUB,~0,1;,~1,
             &SUB,6,&ADD,~1,~2;;;
           MOVE FROM ~1 TO ~2
           &HANOI,&SUB,~0,1;,
             &SUB,6,&ADD,~1,~2;;,~2;>;,
  &DEF,0,;;>;
```

A chaque niveau de récursion

- deux définitions locales sont ajoutées à l'environnement:
  - l'expansion de la première génère deux appels récursifs successifs
  - l'expansion de la seconde génère le string vide
- une de ces deux définitions locales est étendue : la 1ère si l'argument 0(nom) de l'appel est positif, la 2de s'il est nul.

A part l'avantage de sa puissance, ce mécanisme de branchement conditionnel a l'inconvénient d'être très coûteux et fort alembiqué !

2.4.C TRAC

```
&(DS,HANOI,(&(GR,K,0
 (&(HANOI,&(SU,K,1),X,&(SU,6,&(AD,X,Y)))
  MOVE FROM X TO Y
  &(HANOI,&(SU,K,1),&(SU,6,&(AD,X,Y)),Y)
 ),)))   &(SS,HANOI,K,X,Y)
```

2.4.D ML/I

```
MCDEF HANOI , , ; AS
<MCGO LO IF ~A1. LT 1;
 MCSET T1=~A1.-1; MCSET T2=6-~A2.-~A3.;
 HANOI ~T1.,~A2.,~T2.;
 MOVE FROM ~A2. TO ~A3.
 HANOI ~T1.,~T2.,~A3.;>;
```

2.4.E STAGE 2

```
HANOI ',',',
IF e10 LT 1 SKIP 5
e10-1e46                arg4:='arg1-1'
6-e20-e30e56            arg5:='6-arg2-arg3'
HANOI e44,e20,E54
MOVE FROM e20 to e30
HANOI e44,e54,e30
&
```

La récursivité ne pose pas de problème dans aucun de ces langages, vu que l'expansion des appels imbriqués est dynamique. Notons que le coût en est toujours élevé.

2.5 BOUCLE FOR

Supposons que nous désirions exécuter des itérations at macro-time. Pour ce faire, nous écrivons une macrodéfinition 'FOR' de format FOR variable FROM valeur1 [STEP valeur2 ?] UNTIL valeur3 DO instruction ;

2.5.A ASS.

Pour qu'une telle macrodéfinition 'FOR' présente de l'intérêt, il faut que l'argument "instruction" d'un appel puisse contenir une macro-instruction qui puisse être étendue pendant l'expansion de la macro 'FOR'. Comme nous n'avons pas rencontré cette possibilité en ASS, nous passons cet exemple.

2.5.B GPM

Nous nous limitons aux hypothèses suivantes :

- pas = +1 (donc pas de "STEP arg")
- valeurs initiale et finale de la variable contrôlée=entiers positifs
- valeur finale ⩾ valeur initiale

Schéma de la macro-instruction :

```
      &FOR, variable contrôlée,FROM,valeur initiale,UNTIL,valeur finale,
Arguments:  0          1              2        3          4        5
      DO,instruction;
      6       7
```

Macrodéfinition :

```
&DEF,FOR,<&DEF,~1,~3;&~3,~1,~5,~7,
  &DEF,~3,~7<&FOR,~1,FROM,&ADD,~0,1;,
       UNTIL,~2,DO,~3;>;,
  &DEF,&ADD,~5,1;,;;>;
```

Dans un appel à cette macro, l'argument 7 doit être entouré de parenthèses littérales, sinon il est évalué immédiatement... Il est constitué par une expression GPM qui sera évaluée pour chaque valeur de la variable contrôlée. Dans l'exemple qui suit, cette expression est réduite à la variable de contrôle.

Exemple :

```
        &FOR,X,FROM,1,UNTIL,2,DO,<&X;>;
arguments:  0 1  2  3  4   5 6  7
```

L'appel est exécuté comme suit :

- nouvelle définition globale : X⟶1
- création de deux définitions locales au niveau d'expansion 1:
  - 1⟶&X;&FOR,~1,FROM,&ADD,~0,1;,UNTIL,~2,DO,~3;
  - 3⟶vide

  et appel de la macro '1' avec
  - arg1='X'
  - arg2='2'
  - arg3='&X;'

dont l'expansion donne :

- génération de la valeur de X, c'est-à-dire [1]
- appel de la macro 'FOR' avec
  - arg1='X'
  - arg3='2'
  - arg5='2'
  - arg7='&X;'
- nouvelle définition globale : X⟶2
- création de deux définitions locales au niveau d'expansion 2:
  - 2⟶&X;&FOR,~1,FROM,&ADD,~0,1;,UNTIL,~2,DO,~3;
  - 3⟶vide

  et appel de la macro '2' avec
  - arg1='X'
  - arg2='2'
  - arg3='&X;'

dont l'expansion donne :

- génération de la valeur de X, c'est-à-dire [2]
- appel de la macro 'FOR' avec
  - arg1='X'
  - arg3='3'
  - arg5='2'
  - arg7='&X;'

- nouvelle définition globale : X⟶3

- création de deux définitions locales au niveau d'expansion 3 :

$$\begin{cases} 3 \longrightarrow \&X;\&FOR,\sim 1,FROM,\&ADD,\sim 0,1;,UNTIL,\sim 2,DO,\sim 3; \\ 3 \longrightarrow \text{vide} \end{cases}$$

et appel de la macro '3'

La seconde des deux définitions locales '3' est appelée : elle génère un string vide et termine la récursion.

2.5.C TRAC

Nous élargissons les hypothèses de travail :

$$\begin{cases} \text{pas} > 0 \text{ doit être écrit explicitement} \\ \text{valeurs initiale et finale de la variable contrôlée = entiers positifs} \\ \text{valeur finale} \geqslant \text{valeur initiale.} \end{cases}$$

Schéma de la macro-instruction :

```
&(FOR,variable contrôlée,FROM,valeur initiale,STEP,pas,UNTIL,
arguments:          1              2          3          4   5   6
valeur finale,DO,instruction)
      7        8       9
```

Macrodéfinition :

```
&(DS,FOR,(&(DS,X1,X3)                                          (1)
      &(GR,&(X1),X7,,
  (X9&(FOR,X1,FROM,&(AD,X3,X5),STEP,X5,UNTIL,X7,DO,(X9))))))
&(SS,FOR,X1,X2,X3,X4,X5,X6,X7,X8,X9)
```

Dans un appel, l'argument 9 doit être entouré de parenthèses littérales, sinon il est évalué immédiatement.

---

(1) van der Poel W.L. "Advanced Course on Programming Language and data structures" p. 2.8-1.

Exemple:

```
        &(FOR,X,FROM,1,STEP,3,UNTIL,5,DO,(&(X)))
arguments:    1  2  3  4  5  6  7 8  9
```

L'appel est exécuté comme suit :

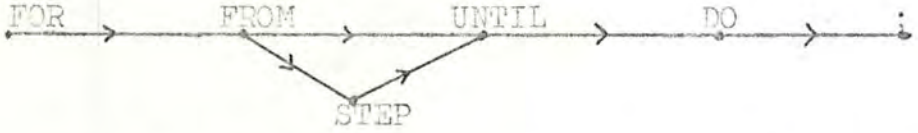
- création de la macrodéfinition globale X⟶1
- appel &(GR,1,5,,(&(X)&(FOR,X,FROM,&(AD,1,3),STEP,3,UNTIL,5,DO,(&(X)))))
- comme 1<5, &(X)&(FOR,X,FROM,&(AD,1,3),....))) est évalué :
- génération de la valeur de 'X', c'est-à-dire [1]
- appel &(FOR,X,FROM,4,STEP,3,UNTIL,5,DO,(&(X))) dont l'expansion se passe comme suit :
- redéfinition de la macro 'X' : X⟶4
- appel &(GR,4,5,,(&(X)&(FOR,X,FROM,&(AD,4,3),STEP,3,UNTIL,5,DO,(&(X)))))
- comme 4<5, &(X)&(FOR,X,FROM,&(AD,4,3),.....))) est évalué :
- génération de la valeur de 'X', c'est-à-dire [4]
- appel &(FOR,X,FROM,7,STEP,3,UNTIL,5,DO,(&(X))) ce qui donne :
- redéfinition de 'X' : X⟶7
- appel &(GR,7,5,,(&(X)&(FOR,X,FROM,&(AD,7,3),STEP,.....))))
- comme 7>5, le string vide est généré et la récursion est terminée.

2.5.D ML/I

Nous étendons les hypothèses de travail comme suit :

- les valeurs initiale et finale de la variable contrôlée et le pas sont des macro-expressions ne contenant pas de variables locales Ti.
- lorsque le pas n'est pas explicité, il vaut +1 (la macro-instruction n'a que 4 arguments au lieu de 5 dans ce cas); sinon, il est positif ou négatif.

La structure de délimiteurs peut se représenter comme suit :



T1:= 4 arguments
T1:= 5 arguments

```
MCDEF FOR FROM OPT STEP N1 OR N1 UNTIL ALL DO ; AS <          (1)
MCSET ∿A1.=∿A2.;          initialisation de la variable contrôlée Pi
MCSET T3=1;               T3 utilisé comme pas, initialisé à 1
MCGO L1 IF T1 EQ 4;       si pas implicite=1, goto∿L1.
MCSET T3=∿A3.             initialisation de T3 si pas ≠ 1
MCGO L1 IF T3 GR 0;       si pas>0, goto∿L1.
∿L2.
MCGO L0 IF ∿A4. GR ∿A1.;  fin lorsque pas < o
∿A5.                      exécuter l'instruction
MCSET ∿A1.=∿A1.+T3;       décrémenter la variable contrôlée
MCGO L2;                  et boucler
∿L1.
MCGO L0 IF ∿A1. GR ∿AT1-1.; fin lorsque pas > 0
∿AT1.                     exécuter l'instruction
MCSET ∿A1.=∿A1.+T3;       incrémenter la variable contrôlée
MCGO L1; >                et boucler
```

Remarque

Dans tout appel, la variable contrôlée doit être une variable permanente Pi. Si c'était une variable temporaire, la macro'FOR' utiliserait ses propres variables locales au lieu d'utiliser les variables de l'environnement appelant! comme ces variables locales, au moins T1 et T3 , sont nécessaires pour exécuter la macro 'FOR', il en résulterait des erreurs imprévisibles !

Exemple:

```
FOR P1 FROM 1 UNTIL 4 DO ∿P1./ ;
        génère     1/2/3/4/

FOR P4 FROM 5 STEP -2 UNTIL 2 DO ∿P1./;
        génère     5/3/
```

(1) Brown P.J. "ML/I User's Manual" p. 7/6

2.5.E STAGE 2

Nous prenons les hypothèses suivantes :

- pas positif ou négatif, explicite
- valeurs initiale et finale de la variable contrôlée et pas sont des expressions arithémtiques

```
FOR ^ FROM ' STEP ' UNTIL ' DO '.
e10 EQU e24              mémoriser valeur initiale de la variable contrôlée
IF e34 LT 0 SKIP 2       si pas < 0
IF e11 GT e44 SKIP 5     fin de boucle pour pas > 0
SKIP 1
IF e11 LT e44 SKIP 3     fin de boucle pour pas < 0
e50                      exécuter l'instruction
e24+e34e26               incrémenter la variable contrôlée
FOR e10 FROM e24 STEP e30 UNTIL e40 DO e50.
&
```

Exemple:

Etant donné la macrodéfinition
```
VAL'.
e11
&
```

```
FOR X FROM 3 STEP -2 UNTIL 2 DO VALX.
```

génère : [3]

```
FOR X FROM 1 STEP -2 UNTIL 2 DO VALX.
```

qui ne génère rien et termine la récursion.

STAGE2 a donc le même inconvénient que GPM et TRAC: vu qu'il n'y a pas de branchement "en arrière", il faut brancher par appel imbriqué, ce qui coûte cher !

Pour éviter cela, nous pouvons calculer le nombre d'itérations en début d'expansion en utilisant la fonction eF7. Cela donne :

```
nombre d'itérations:= (valeur finale - valeur initiale + pas)/pas
FOR ' FROM ' STEP ' UNTIL ' DO '.
(e44-e24+e34)/e34e96              arg9:='nombre d'itérations'
e24-e34e86                        arg8:='valeur initiale-pas'
e10 EQU e84             variable contrôlée:=valeur initiale-pas
e94eF7                  début d'itération
e11+e34e86              arg8:='valeur courante de variable contrôlée+pas'
e10 EQU e84             variable contrôlée:=sa valeur courante+pas
e50                     exécution de l'instruction
eF8                     fin de la portée de l'itération
&
```

Etant donné la macrodéfinition VAL'.

```
e11
&
```

si la variable globale N vaut 5 au moment de l'appel,

FOR X FROM N+2 STEP -N+1 UNTIL 3 DO VALX.

donne:

- arg9:=(3-7+-4)/-4 = 2
- arg8:=7--4 = 11
- X :=11
- début d'itération : nombre d'itérations:=2 (valeur de arg9)
- 1ère itération: arg8:=11+-4
  X:=7
  VALX génère | 7 |
- 2de itération : arg8:=7+-4
  X:=3
  VALX génère | 3 |
- fin d'expansion

Dans le cas où le pas est implicitement 1, une seconde macrodéfinition doit être écrite :

```
FOR ' FROM ' UNTIL ' DO '.
e34-e24+1e96
e24-1e86
e10 EQU e84
e94eF7
e11+1e86
e10 EQU e84
e50
eF8
&
```

ML/I semble donc de loin le macrolangage le plus souple pour un tel type de macrodéfinitions.

2.6 Macrogénération différée

Nous avons vu l'intérêt de directives de macrogénération différée(cfr. 1.4). Nous proposons ici des macrodéfinitions permettant d'insérer du texte généré à un endroit du texte cible qui est spécifié dans le texte source.

Nous illustrons l'emploi de ces macros au moyen d'un exemple simple :

étant donné une macro A:=B+C

1.- générer du code pour exécuter ces opérations à l'endroit de l'expansion

2.- générer les réservations de constantes en fin du texte cible (pour simplifier, nous supposons que les arguments de toutes les macro-instructions A:=B+C sont différentes).

2.6.A XPOP

Comme ce macro-assembleur possède des pseudo-instructions de macrogénération différée très souples (cfr. 1.4), nous proposons un exemple XPOP (à notre connaissance, le MACRO-ASSEMBLER 360 ne possède pas de telles directives).

Soit la macrodéfinition :

```
ASSIGN   MACRO   C,A,B
         CLA     A
         ADD     B
         STO     C
         WAIT
A        RESERVE 1
B        RESERVE 1
C        RESERVE 1
         END
```

Le texte

```
ASSIGN   Z,X,Y
ASSIGN   K,I,J
```

génère

```
        CLA     X
        ADD     Y
        STO     Z
        CLA     I
        ADD     J
        STO     K
X       RESERVE 1
Y       RESERVE 1
Z       RESERVE 1
I       RESERVE 1
J       RESERVE 1
K       RESERVE 1
```

2.6.B ML/I

```
MCSET P1=0;
MCDEF REMOTE ; AS< MCSET P1=P1+1;
        MCDEFG I∿P1. AS ∿A1.; > ;
MCDEF INSERT AS <
  FOR P2 FROM 1 UNTIL P1 DO COPY I∿P2./;>;
MCDEF COPY / AS <
     MCDEF TEMP AS ∿A1.; TEMP > ;
```

(1)

Exemple:

```
        MCDEF ASSIGN :WITH= + ; AS <
            LOAD    ∿WA2.
            ADD     ∿WA3.
            STORE   ∿WA1.
        REMOTE ∿WA1.  RES    1;
        REMOTE ∿WA2.  RES    1;
        REMOTE ∿WA3.  RES    1;> ;
```

---

(1) Brown P.J. "ML/I User's Manual" p. 7/12

Le texte ASSIGN Z:=X+Y;
ASSIGN K:=I+J;
INSERT

est évalué comme suit

- génération de LOAD X
  ADD Y
  STORE Z

- appel 'REMOTE X RES 1;'

  d'où création de deux liaisons globales :

  { P1 ⟶ 1 (variable permanente)
  { I1 ⟶ X RES 1 (macro globale)

- appel 'REMOTE Y RES 1;'
  d'où { P1 ⟶ 2 (mise à jour de la variable permanente)
  { I2 ⟶ Y RES 1

- appel 'REMOTE Z RES 1;'
  d'où { P1 ⟶ 3
  { I3 ⟶ Z RES 1

- génération de LOAD I
  ADD J
  STORE K

- appel 'REMOTE I RES 1;'
  d'où { P1 ⟶ 4
  { I4 ⟶ I RES 1

- appel 'REMOTE J RES 1;'
  d'où { P1 ⟶ 5
  { I5 ⟶ J RES 1

- appel 'REMOTE K RES 1;'
  d'où { P1 ⟶ 6
  { I6 ⟶ K RES 1

- appel de 'INSERT' , d'où

- appel 'FOR P2 FROM 1 UNTIL P1 DO COPY I↷P2./;'

- 1ère itération : appel 'COPY I~P2./' (cfr.remarque ci-dessous)
  d'où création de la liaison

        TEMP ⟶ X RES 1 (resultat de l'évaluation de I1)

  et appel ' TEMP' qui génère

        X RES 1

-2ème à 6ème itération : génération de

        Y RES 1
        Z RES 1
        I RES 1
        J RES 1
        K RES 1

Nous pouvons utiliser un mécanisme analogue dans les autres macro-langages.

A noter que les strings générés sont mémorisés comme macrodéfinitions globales, ceci afin de pouvoir les insérer dans le texte cible à n'importe quel niveau d'expansion du texte source.

Remarque

Le recours à une macrodéfinition locale 'TEMP' est nécessaire pour pouvoir générer dynamiquement les appels 'I1','I2',.... dont les expansions délivrent les textes générés à insérer. Sans cela, nous ne pourrions générer que les noms 'I1','I2',... et non leurs valeurs! Nous pourrions nous passer de la macro 'COPY', et écrire uniquement la macro :

```
MCDEF INSERT AS <
  FOR P2 FROM 1 UNTIL P1 DO
  MCDEF TEMP AS I~P2.; TEMP  ; > ;
```

Il en résulterait cependant une perte de place mémoire pendant l'expansion de la macro 'FOR', puisque P1 macrodéfinitions locales 'TEMP'seraient ajoutées au lieu d'une seule comme dans la solution proposée.

2.6.C GPM

En GPM comme en TRAC, le fait que les arguments sont passés par valeur immédiate permet la génération dynamique des appels 'I1','I2',...

```
&DEF,PX,000;
&DEF,REMOTE,<&UPDATE,PX,&ADD,&PX;,1;;
             &DEF,I&PX;,~1;>;
&DEF,INSERT,<
  &FOR,AA,FROM,1,UNTIL,&PX;,DO,<&VAL,I&AA;>;>;
```

2.6.D TRAC

```
&(DS,PX,0)
&(DS,REMOTE,(&(DS,PX,&&(AD,&&(PX),1))
             &(DS,I&(PX),X1)))
  &(SS,REMOTE,X1)
&(DS,INSERT,(&(FOR,AA,FROM,1,BY,1,UNTIL,
     &(PX),DO,(&(I&(AA))&(DD,I&(AA))))))
```

TRAC est le seul macrolangage qui permet de supprimer les liaisons globales et donc de récupérer la place mémoire qu'elles occupaient.

2.6.E STAGE 2

```
REMOTE '.
RRe26              arg2:=RR
e21+1e26           arg2:='valeur de RR+1'
RR EQU e24         incrémentation de la valeur de la variable RR
PPe24 EQU e10      PPi:=copie littérale de arg1
&

INSERT.
FOR AA FROM 1 UNTIL RR DO COPY AA.
&

COPY '.
PPe11e26           arg2:='PPi'
e21
&
```

Avant d'utiliser REMOTE dans le texte de génération, il faut initialiser RR à 0 par une macro-instruction

RR EQU 0.

3. CONCLUSION : TENTATIVE D'EVALUATION DES TECHNIQUES DE MACROGENERATION

3.1 Introduction

Lorsque nous tâchons d'évaluer la qualité d'un outil de traitement de l'information (un macrogénérateur en est un cas particulier),nous pouvons entre autres distinguer trois aspects que nous appelons puissance, souplesse et efficience.

La puissance indique quels traitements peuvent être réalisés à l'aide de l'outil en question, traitement étant envisagé sous l'angle de la relation données-résultats (texte source - texte cible). Dans ce premier aspect, nous nous intéressons donc uniquement aux fonctions qui peuvent être définies et exécutées au moyen de l'outil, non à la manière dont elles le sont.

La souplesse est une estimation de l'aisance avec laquelle les relations données-résultats peuvent être décrites par l'utilisateur de l'outil. Dans ce second aspect, nous envisageons donc la manière dont l'outil peut être utilisé par un être humain.

L'efficience est une mesure des performances de l'outil considéré. Elle exprime le coût (temps d'exécution, place mémoire, ...)résultant de son utilisation. Dans ce troisième aspect, nous considérons donc la manière dont les fonctions décrites au moyen de l'outil sont (ou peuvent être) exécutées par l'ordinateur.

Nous pouvons recourir à plusieurs théories (théorie des automates, théorie de la récursivité, ...) pour déterminer la puissance d'un outil [(1)] Par exemple, il est généralement possible de montrer qu'un langage déterminé permet de simuler toute machine de Turing, et que dès lors ce langage permet de décrire toute fonction dite partiellement calculable. On se rend ainsi compte que la puissance des macrogénérateurs est généralement très grande. Nous n'insisterons donc pas sur cet aspect.

Du point de vue souplesse, par contre, l'estimation est beaucoup plus arbitraire. La souplesse concerne en effet "l'aisance" avec laquelle un outil peut être manipulé par un être humain. Pour mesurer cette"aisance", il faudrait pouvoir définir un référentiel. Comme nous ne possédons

---

(1) Notre définition de la puissance devrait être affinée de façon à tenir compte du jeu de caractères sur lequel cette puissance peut être atteinte. Nous considérons brièvement ce point en 3.2.A.2 (indépendance de notation).

actuellement pas de critères permettant de décrire avec exactitude les schémas mentaux humains, il n'est guère possible d'en définir de façon à la fois précise et réaliste.

Il faut à notre avis se garder de remplacer la notion floue de souplesse que nous tentons d'introduire par une notion de "souplesse théorique" qui serait beaucoup plus précise ... mais irréaliste. Ainsi, si nous convenions qu'un outil est d'autant plus souple qu'il permet d'exprimer les mêmes choses de façon plus synthétique au moyen d'un nombre minimum de mécanismes, il serait possible de mesurer cette souplesse de manière plus précise. Mais une telle démarche risque de conduire à un double défaut :

- des expressions très synthétiques ne sont pas nécessairement naturelles .... parfois elles risquent même de ne l'être pas du tout! La "souplesse théorique" n'implique donc pas la souplesse telle que nous l'entendons.

- il y a de fortes chances que des outils offrant une grande souplesse théorique soient très inéfficients une fois qu'ils sont implémentés, compte tenu de la structure actuelle des ordinateurs.

Un exemple de ce danger est entre autres fourni par TRAC (et encore davantage par GPM) : à l'aide d'un très petit nombre de mécanismes, très simples dans leurs principes, ce macrolangage permet de décrire énormément de fonctions. En particulier, le dynamisme de création, de modification et d'appel des macrodéfinitions offre une souplesse théorique énorme (une macrodéfinition peut être utilisée comme variable de macrogénération, comme branchement, ...). Mais sur le plan pratique,ces mécanismes exigent une gymnastique d'esprit qui frise l'acrobatie (voir e.a. exemple 2.3.C). De plus, l'efficience des implémentations de TRAC est très faible. Des outils beaucoup moins généraux, beaucoup moins "élégants" du point de vue théorique, sont souvent plus simples à utiliser et coûtent beaucoup moins cher en temps d'exécution et en place mémoire. C'est le cas des directives de transfert du type AIF et du mécanisme de variables de macrogénération différencié de celui des macrodéfinitions.

Tout ce que nous pouvons dès lors faire, dans l'état actuel des choses, c'est d'envisager certains problèmes types des techniques de macrogénération.Nous examinons comment ces problèmes sont résolus dans les macrolangages existants. A partir de cet examen, nous pouvons comparer la facilité d'emploi des différents macrolangages.

3.2 Les macro-instructions

La macro-instruction est sans doute la structure syntaxique la plus fondamentale des macrolangages, à tel point que dans certains d'entre eux, les directives de macrogénération et de manipulation de liaisons symbole-valeur n'en sont que des cas particuliers.Les conventions adoptées au niveau des macro-instructions déterminent donc en grande partie la "qualité" des macrogénérateurs.

3.2.A Reconnaissance des macro-instructions

Par reconnaissance d'une macro-instruction, nous entendons ici sa délimitation, l'identification de la macrodéfinition appelée et la séparation de ses arguments.

3.2.A.1 Efficience des algorithmes de reconnaissance

L'efficience dépend d'une part de la manière dont la macro appelée est identifiée, d'autre part de la manière dont les arguments sont séparés.

a.- identification par nom : dans ce cas, le recours à une "hash function" rend l'identification de la macro appelée très efficiente. Rappelons néanmoins la solution adoptée en GPM: la table des macrodéfinitions est organisée sous forme d'une chaîne parcourue en sens inverse à partir de la dernière entrée. A chaque instant, le nombre moyen d'accès requis pour déterminer la macro appelée est approximativement égal à n/2, où n désigne le nombre de macrodéfinitions présentes dans l'environnement! Une telle solution est évidemment très coûteuse.

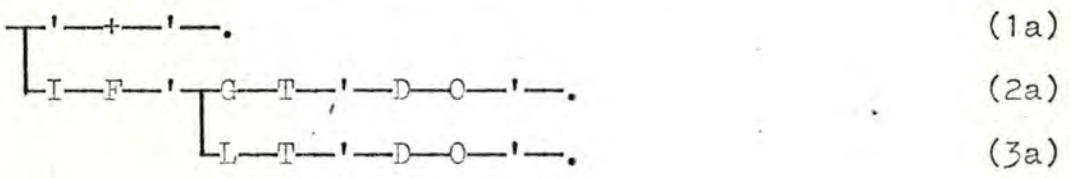
Dans les cas où les séparateurs d'arguments sont des symboles prédéfinis (ASS, GPM, TRAC), l'algorithme de reconnaissance est simple et efficient. Il n'en va plus de même en ML/I, où les délimiteurs-définis par l'utilisateur- sont mémorisés dans un arbre. En effet, puisque chaque structure de délimiteurs représente en fait un graphe orienté, la ligne prototype de chaque macrodéfinition est mémorisée sous forme d'un arbre. Chaque atome d'une macro-instruction doit donc être comparé aux successeurs possibles du délimiteur le plus récemment identifié , jusqu'à ce que le délimiteur terminal soit reconnu.

Ce processus est beaucoup moins efficient que celui de GPM, TRAC, ... mais la souplesse de la syntaxe des macro-instructions qui en résulte (cfr. 3.2.A.2) peut justifier son coût. Notons qu'en ML/I, une macro-instruction est identifiée dès qu'un atome du texte source est identique à un nom de macro, et que le texte d'une macro-instruction est scanné de gauche à droite sans retour en arrière (ce qui n'est pas le cas de LIMP et STAGE2). Il en résulte que l'oubli d'un délimiteur dans la rédaction d'une macro-instruction peut entraîner une fin erronée de macrogénération!

b.- identification par nom distribué (LIMP, STAGE2) : les clichés des macrodéfinitions sont mémorisés sous forme d'arbre. Ainsi l'ensemble des clichés

'+'. (1)
IF'GT'DO'. (2)
IF'LT'DO'. (3)

est mémorisé comme suit en STAGE2 :



(1a)
(2a)
(3a)

L'organigramme de l'algorithme de template matching est reproduit dans l'annexe. A noter que, lors de leur évaluation, chaque ligne du texte de génération et chaque ligne - non terminée par 'eF1'-appartenant à un corps est scanné, avec éventuellement des retours en arrière, jusqu'à ce qu' elle rencontre un cliché. Si elle n'en rencontre aucun, elle est recopiée dans le texte cible.
Pour donner une idée de la lenteur de l'algorithme, indiquons brièvement ce qui se passe lorsque STAGE2 évalue la ligne 'IF1LT2THEN3.' si la table de macrodéfinitions contient les trois clichés précédents :

le parcours de l'arbre donne d'abord

cliché (3a) avec 'arg1→1' et 'arg2 →2THEN3'

ensuite, puisque le cliché (3a) ne rencontre pas cette ligne,

cliché (2a) avec 'arg1 →1LT2THEN3'

ensuite, puisque le cliché (2a) ne peut être retenu,

cliché (1a) avec 'arg1 →IF1LT2THEN3'

enfin, puisque le cliché (1a) ne satisfait pas et que tous les clichés ont été"essayés", la ligne est sortie !

Le coût d'un tel processus est prohibitif : l'algorithme de template matching est encore beaucoup plus lent que celui de ML/I. En effet, chaque ligne est comparée à l'ensemble des clichés, avec d'éventuels retours en arrière, alors qu'en ML/I , un cliché est identifié dès que le nom (délimiteur 0) d'une macro est reconnue et le scanning se fait exclusivement de gauche à droite. Waite (1) justifie l'utilité de STAGE2 malgré la lenteur en faisant remarquer :

"STAGE2 does not provide all features which one would like to see in a macro processor, it is relatively slow, and it requires a fair amount of data space. Its purpose is to provide a common macro processor for realising a variety of abstract machines, and not to act as a processor for day-to-day use by applications programmers. For this latter use, we are preparing versions of ML/I and LIMP".

Pour pouvoir utiliser"quotidiennement" STAGE2 de manière assez efficiente, il faudrait introduire soit des restrictions sur la syntaxe des clichés, soit des conventions permettant de rendre l'algorithme plus rapide (par exemple en subdivisant l'arbre des macrodéfinitions en sous-arbres).

---

(1) W.M. Waite "Building a mobile programming system", Computer Journal, vol.13, February 1970, p.30.

3.2.A.2 Puissance résultant de l'indépendance de notation

Nous avons dit que la puissance des macrolangages étudiés est équivalente. La puissance ainsi définie traduit le fait que ces langages permettent de décrire toute fonction dite partiellement calculable. Du point de vue pratique, ce critère est insuffisant. Il ne tient en effet aucun compte de la représentation des arguments ni de celle du résultat. Il est possible de définir une puissance relativement à un alphabet donné. Sans nous étendre sur cet aspect, montrons sur un exemple que la notion de puissance ainsi définie correspond mieux à ce que l'on peut en attendre sur le plan pratique.

Dans des macrolangages comme ASS, GPM, et TRAC, plusieurs caractères prédéterminés - les marqueurs - ont des fonctions spécifiques. Il en résulte que les textes de génération ne peuvent pas être écrits "librement" : leur notation est dite dépendante. Cela supprime toute possibilité de produire des textes cible à partir de textes de génération à format "libre". Ce n'est pas le cas de LIMP , ML/I et STAGE2. Leur grande indépendance de notation élargit leur champ d'applications. Elle offre entre autres la possibilité de produire des textes à partir de textes de génération écrits dans n'importe quel langage évolué.. Nous constatons ainsi que l'indépendance de notation au niveau des textes de génération accroît la "puissance pratique" d'un macrogénérateur.

3.2.A.3 Souplesse et délimitation des macro-instructions

Dans tous les macrolangages étudiés ici, une macro-instruction se réduit à un string dont la délimitation (début et fin) est déterminée de manière standard. La levée de cette restriction permettrait une souplesse plus grande, mais au prix d'une efficience encore moindre.

Ainsi, la définition de conventions de délimitation plus générales dans des macrolangages comme ML/I et STAGE2 (possibilité de spécifier la syntaxe d'une partie de macro-instruction qui précéderait le nom en ML/I, possibilité de commencer une macro-instruction à n'importe quel endroit du texte source en STAGE2,..) semble utopique dans l'état actuel des choses. En effet, d'une part elle rendrait l'algorithme de reconnaissance incroyablement lent, d'autre part elle risquerait d'être la source d'ambiguïtés

insurmontables.
La seule extension entreprise jusqu'ici dans ce domaine concerne des "special purpose macrogenerators" dont le langage de base est un langage évolué. Un tel macrogénérateur permet à l'utilisateur de spécifier d'une part à quelle classe syntaxique du langage de base un argument d'un appel doit appartenir, d'autre part la ou les structures syntaxiques du langage de base dans lesquelles une macro-instruction peut apparaître. Cette dernière possibilité généralise les conventions de délimitation des appels. Nous ne nous attarderons pas plus à ces techniques : les problèmes qu'elles soulèvent dépassent le cadre de cette étude, tant au plan de la définition des critères que de l'efficience. Remarquons néanmoins qu'elles requièrent l'action conjointe du macrogénérateur et d'un analyseur syntaxique. Le texte source n'est donc plus scanné comme une simple chaîne de caractères, mais comme un texte dont la syntaxe doit être conforme à la grammaire du langage de base.

3.2.B Passation des arguments

Nous avons déjà souligné les principaux avantages et inconvénients des passations à priori et à posteriori (appel par valeur immédiate et par nom) en 1.3.B.

A ce point de vue, ML/I n'a pas l'uniformité des autres macrolangages : les arguments des macros-système sont passés par valeur immédiate, ceux des macros-utilisateur le sont par nom. Cette irrégularité est nécessaire pour pemettre, d'une part la mémorisation correcte des structures de délimiteurs (prise en charge des caractères "espace" et "newline"), d'autre part, la génération dynamique de délimiteurs et donc de macro-instructions. L'exemple 2.6.B montre que ce n'est sans doute pas la solution la plus souple pour générer dynamiquement des macro-instructions.

Une amélioration éventuelle, quoique inexistante, serait d'adjoindre aux macrolangages des conventions permettant à l'utilisateur lui-même de spécifier le type de passation des arguments d'une macro-instruction.

3.2.C Rescan du texte généré

Nous avons vu qu'en TRAC, LIMP et STAGE2 (cfr. 1.3.B,b) l'évaluation d'une macro-instruction imbriquée dans un texte de remplacement se fait en deux étapes : construction de la macro lors d'un premier scanning, identification de la macro appelée et passation des arguments lors du rescan. La présence du rescan permet de générer dynamiquement des appels en construisant les délimiteurs pendant le premier scan. Cette solution est plus souple que celle proposée ci-dessus. Elle a l'inconvénient assez grave de provoquer systématiquement deux scannings, ce qui entraîne une perte d'efficience inutile pour l'avantage qu'on peut en retirer.

3.3 Variables de macrogénération

3.3.A Macrodéfinitions dynamiques

Un premier avantage résultant du dynamisme des macrodéfinitions est la possibilité de créer de nouvelles définitions at macro-time.

D'autre part, nous avons déjà vu que ce mécanisme est assez puissant pour permettre la manipulation de variables de macrogénération: GPM et TRAC ne connaissent que ce type de macro-variables. Les exemples du chapître 2 montrent combien cela peut réduire la souplesse dès que le nombre de macro-variables est élevé. De plus, comme la valeur de telles variables est délivrée par macro-instruction, l'efficience de ce mécanisme de macro-variables est liée à l'efficience de la reconnaissance des macro-instructions. GPM est donc très inefficient à ce point de vue.

Enfin, la limitation à ce seul mécanisme des macrodéfinitions dynamiques réduit le type des macro-variables au seul type string, ce qui diminue encore l'efficience.

3.3.B Types des macrovariables

Les variables de type string suffisent à tous les besoins, mais à un prix plus élevé. Pour éviter les nombreuses conversions string-décimal ou string-binaire, il est donc souhaitable qu'un mécanisme de variables de type entier (mémorisées dans le code binaire ou D.C.B. de l'ordinateur) soit présent.

3.3.C Accès aux liaisons locales

Nous avons fait remarquer en 1.1.C.3 qu'aucun macrogénérateur ne permet l'accès aux liaisons locales de niveau inférieur, ce qui peut étonner, vu que ces liaisons sont présentes dans l'environnement. Il serait peut-être intéressant d'introduire une structure de blocs dans les macrolangages. Du point de vue efficience, il n'en résulterait peut-être pas de grands changements, vu que les liaisons de niveau inférieur sont de toute façon dans l'environnement. Pour communiquer des informations d'une macro-expansion à une autre, il ne serait plus toujours nécessaire soit de recourir à des liaisons globales, soit de passer les valeurs comme arguments. De plus, il serait possible de modifier au niveau i une variable locale créée à un niveau $j < i$, ce qui n'est possible que pour des liaisons globales dans les macrogénérateurs étudiés ici.

De ces considérations sur les variables de macrogénération, il résulte qu'un mécanisme de variables de type entier et de type string différent du mécanisme des macrodéfinitios est souhaitable. Ainsi, la combinaison des variables entières de ML/I et des variables de type string de STAGE2 donnerait une souplesse et une efficience excellente à ce point de vue.

3.4 Directives de macrogénération

3.4.A Directives de calcul

Comme tous les calculs at macro-time sont interprétatifs, il en résulte une certaine inefficience.

3.4.A.1 Arithmétique

Il est souhaitable que les macrolangages possèdent des directives de calcul d'expressions arithmétiques et d'assignation, comme ASS et ML/I. Les directives à deux opérandes de GPM et TRAC manquent en effet de souplesse.

3.4.A.2 Manipulation de strings

L'exemple 2.3.C donne une idée de la souplesse qui peut résulter de directives puissantes de manipulation de string. Etant donné la structure des ordinateurs actuels, l'exécution de telles directives demande beaucoup de temps et diminue donc l'efficience. Leur présence dans les macrolangages reste néanmoins souhaitable, vu les possibilités qui en résultent (la comparaison des exemples 2.3.B et 2.3.C suffit pour s'en rendre compte). Mais l'adjonction d'autres mécanismes permet de gagner sur le plan de la souplesse et de l'efficience.

3.4.B Directives de transfert

Tous les macrolangages recourent au transfert de contrôle par appel imbriqué, mais seuls GPM et TRAC s'y limitent. Il en résulte une grande inefficience, due au fait que les branchements par appel imbriqué ajoutent chaque fois un niveau au sommet du stack de macro-expansion: d'où perte de temps due à la mise à jour des pointeurs de stack et perte de place assez considérable (les exemples 2.5.B et 2.5.C sont significatifs à ce point de vue). De plus, la limitation à ce seul type de branchement entraîne un manque de souplesse pour indiquer des macrogénérations conditionnelles (voir les exemples GPM et TRAC du chapître 2). Nous voyons ici un exemple de mécanisme offrant une grande "souplesse théorique". La structure même de ce mécanisme ne nous permet cependant pas d'en exploiter toutes les possibilités.

La solution adoptée par STAGE2 (comptage des lignes à sauter) empêche les sauts en arrière, ce qui est un inconvénient assez lourd (il suffit de comparer les exemples 2.5.D et 2.5.E pour s'en rendre compte). De plus, comme les fonctions-système de STAGE2 ne peuvent être utilisées qu'en créant des macros ad hoc (cfr. 2.1.E), il en résulte en plus une ineffficience due à la lenteur du processus de reconnaissance des macro-instructions en STAGE2.

La meilleure solution semble donc être celle de ASS et ML/I: directives explicites de type goto. Cependant, il est utile de remarquer que l'efficience de ces directives dépend de la manière dont les étiquettes de macrogénération (apparaissant dans les textes de remplacement) sont implémentées dans la table des macrodéfinitions.

3.4.C Macrogénération différée

La solution de XPOP est sans aucun doute la meilleure. Dans tous les autres cas (excepté TRAC), une perte de place résulte du fait que les textes générés mémorisés pour insertion ultérieure dans le texte cible appartiennent à l'environnement global : leur portée est donc systématiquement la durée de macrogénération.

L'introduction de directives explicites de macrogénération différée semble souhaitable : si elles sont bien conçues, elles entraînent une perte de place minimale et une plus grande souplesse.

3.5 Lisibilité du texte source

Une lecture attentive des exemples du chapître 2 montre que la lisibilité des textes de remplacement - qui constitue un des facteurs de souplesse - varie d'un macrolangage à l'autre. Elle dépend de la manière dont les structures du macrolangage correspondent à nos structures mentales, et ne peut donc être mesurée dans l'état actuel des choses.

Nous pouvons cependant constater que, si la lecture d'une macrodéfinition STAGE2 un tant soit peu complexe est difficile au début, il est néanmoins assez aisé de s'habituer à la notation simple et uniforme des conversions de paramètre et fonctions-système, qui représentent chacune une action simple du système . Il nous semble que nous ne pouvons pas en dire autant de langages comme TRAC et GPM. La définition de directives de calcul et de transfert facilement identifiables constitue certainement une aide à ce point de vue.

B I B L I O G R A P H I E

McILROY, M.D.:"Macro-instruction extensions of compiler languages", Comm. ACM, vol.3, April 1960, pp. 214-220

BROWN, P.J.:"A survey of macro-processors", Annual Review in Automatic Programming, Pergamon Press, 1969, pp. 37-88

CAMPBELL-KELLY, W.:"An introduction to macros", Macdonald/American Elsevier Computer Monographs, 21, 1973

WEGNER, P.:"Programming languages, information structures and machine organization", McGraw Hill, 1971, pp. 130-180

GRIES, D.:"Compiler construction for digital computers", John Wiley & Sons, 1971, pp. 412-434

ASSABGUI, M.:"Le langage d'assemblage : l'assembleur OS/360", Dunod Université, 1972, pp. 152-180.

HALPERN, M.I.:"XPOP : a meta-language without metaphysics", Proc. AFIPS 1964 Fall Joint Computer Conference, vol. 26, pp. 57-68

STRACHEY, C.:"GPM:a general purpose macrogenerator", Computer Journal, vol. 8, October 1965, pp. 225-241

MOOERS, C.N.:"TRAC, a procedure describing language for the reactive typewriter", Comm. ACM, vol. 9, March 1966, pp. 215-219

VAN DER POEL, W.L.:"Advanced course on programming languages and data structures", Amsterdam 1972, chapter 2 (the programming language TRAC)

WAITE, W.M.:"LIMP : a language independent macro processor", Comm. ACM, vol. 10, July 1967, pp. 433-440

BROWN, P.J.:"The ML/I user's manual", 3d edition, University Mathematical Laboratory, Cambridge, 1967

BROWN, P.J.:"The ML/I macro processor", Comm. ACM, vol 10, October 1967, pp. 618-623

POOLE, P.C. and WAITE, W.M.:"The STAGE2 macroprocessor user reference manual", U.K. Atomic Energy Authority, Culham Laboratory, Abingdon Berkshire, 1970

ORGASS, R.J. and WAITE, W.M.:"A base for a mobile programming system", Comm. ACM, vol 12, September 1969, pp. 507-509

WAITE, W.M.:"Building a mobile programming system", Computer Journal, vol. 13, February 1970, pp. 28-31

BROWN, P.J.:"Using a macro processor to aid software implementation", Computer Journal, vol. 12, November 1969, pp. 327-331

LEAVENWORTH, B.M.:"Syntax macros and extended translation", Comm. ACM, vol. 9, November 1966, pp. 790-793

CHEATHAM, T.E.:"The introduction of definitional facilities in higher level programming languages", Proc. AFIPS 1966 Fall Joint Computer Conference, vol. 29, pp. 623-637

LEROY, H.:"A macro-generator for ALGOL", Proc. AFIPS 1967 Spring Joint Computer Conference, vol. 30, pp. 663-669

GALLER, B.A. and PERLIS, A.J.:"A proposal for definitions in ALGOL", Comm. ACM, vol. 10, April 1967, pp. 204-219

HALPERN, M.I.:"Toward a general processor for programming languages", Comm. ACM, vol. 11, January 1968, pp. 15-25

A N N E X E

IMPLEMENTATION D'UN SUBSET DE STAGE2.

Au début de ce travail, avant d'aborder l'étude comparative de macrogénérateurs, nous avons jugé bon d'en examiner un de plus près en l'implémentant. Ceci nous a permis de cerner certains problèmes liés aux techniques de macrogénération en les abordant d'un point de vue pratique.

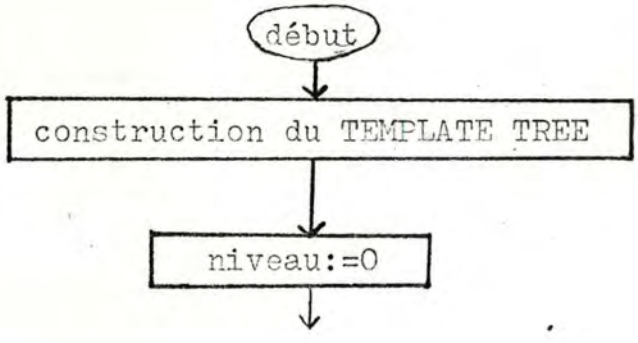
Pendant le stage à l'Université de Warwick, nous avons donc implémenté un subset de STAGE2 sur un ordinateur MODULAR ONE. Cette implémentation est rédigée en BCPL. Notre but n'était pas de créer une implémentation complète et efficiente de ce macrolangage : nous avons donc écarté ou restreint certains aspects.

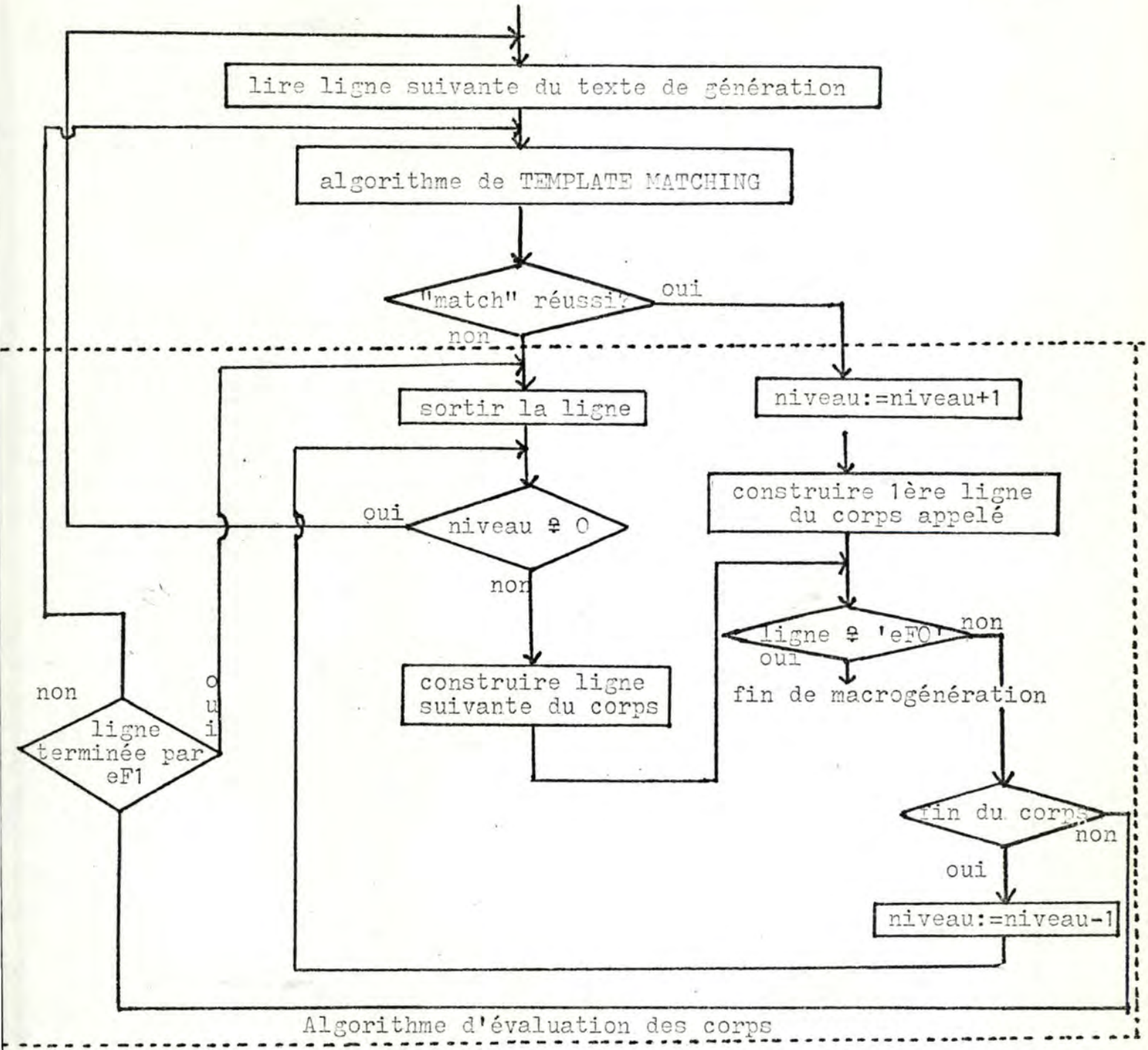
Une brève présentation de l'organisation du programme précède le listing. Les notations utilisées dans cette présentation sont celles de BCPL, cfr. "The BCPL Programming Manual" by M. Richards, Warwick version edited by M.D. May.

L'implémentation a été réalisée à partir de la définition du macrolangage STAGE2, cfr.

"The STAGE2 macroprocessor user reference manual" by P.C. Poole and W.M. Waite.

1. Flowchart





2. Programme principal et sous-programmes

- Programme principal (dans le fichier/STAMA):
  actions : 1.- réservation de mémoire et initialisations
  2.- construction du Template Tree
  3.- appel de la procédure TEMMAT

- Procédure TEMMAT : deux parties :
  TEMMA : algortihme de Template Matching
  COBSCA: algorithme d'évaluation des textes de remplacement

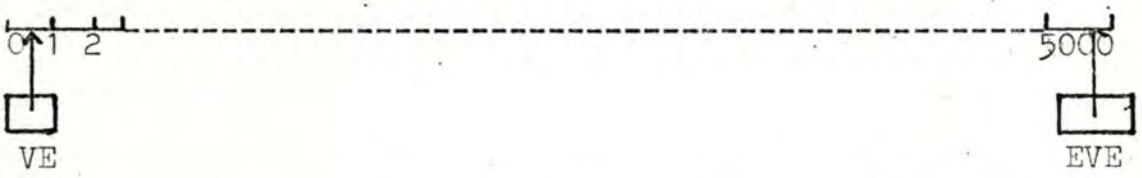
NOTE : la fonction système 'eF0' indique la fin de la macrogénération.

- Procédures appelées pendant l'évaluation des corps :
  - conversions de paramètres: PARSY(symbole généré), PARO, PAR12, PAR3, PAR4, PAR5, PAR6, PITER(itérations), PAR8
  - fonctions-système: FUN1, FUN2, FUN3, FUN4, FUN5, FUN6, PITER, FUN9
- Procédures utilisées par les conversions de paramètre et les fonctions-système :
  - recherche d'un argument : SCH
  - table des symboles : PACK, EXFREE, SRCH, VALID, CRID, ASSID
  - calcul d'expressions arithmétiques : DEC, BIN, NUM, ARITH
  - transferts : SKIP
- Procédures d'entrée sortie :
  - REAFLA : lecture de la flag line
  - READIN : lecture d'une ligne du texte de génération
  - WRERR : messages d'erreur
  - WRILI : écriture d'une ligne du texte cible

3. Réservations de mémoire et initialisations

a.- 66 symboles globaux (marqueurs, pointeurs, ...)

b.- espace de travail : vecteur de 5001 mots (mots de 16 bits)

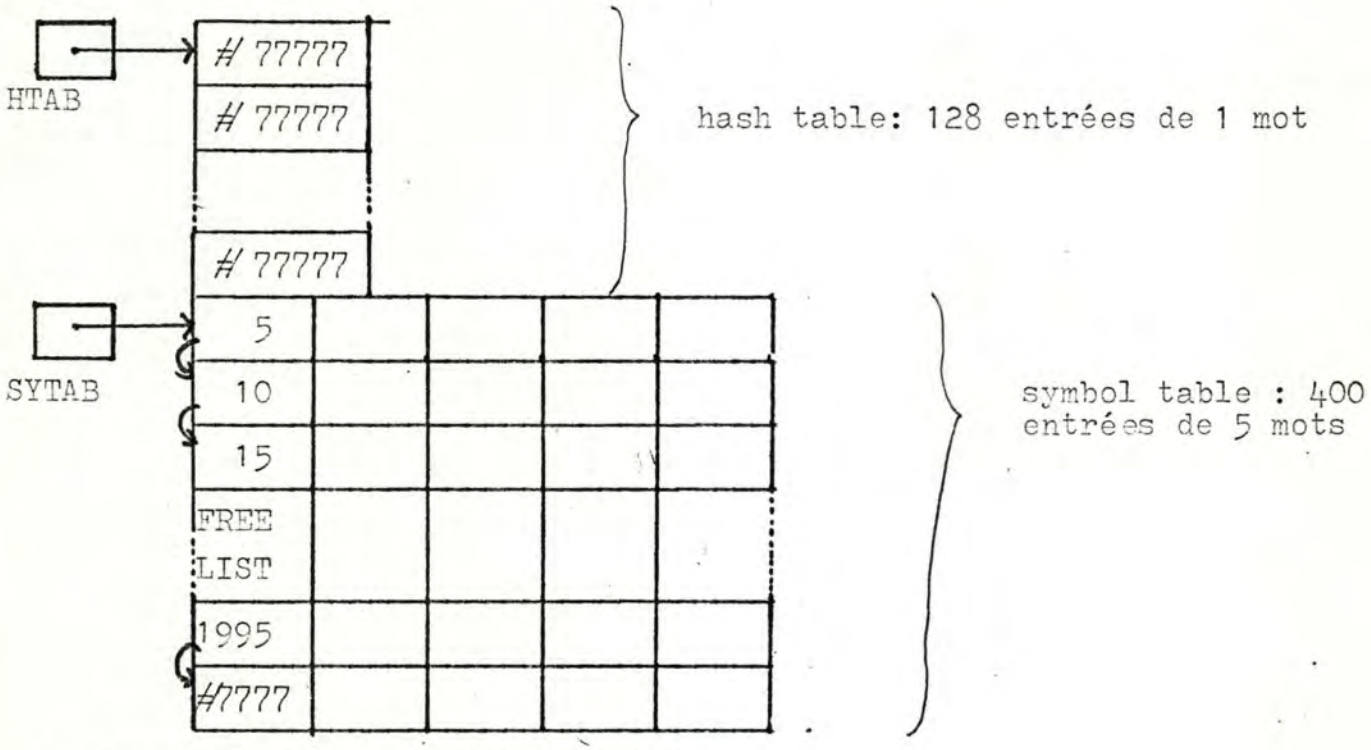


Les globaux VE et EVE pointent (adresse absolue) vers le premier et le dernier mot de ce vecteur

c.- table des symboles : vecteur de 2.128 mots

Note

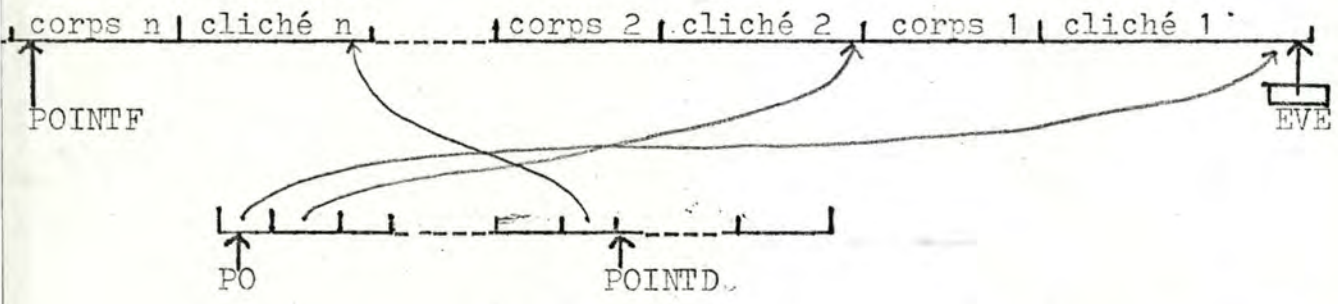
précède une constante locale



4. Construction du "TEMPLATE TREE"

Nous procédons en trois étapes :

1.- lecture séquentielle des macrodéfinitions en fin de vecteur VE.
Lorsque toutes les macrodéfinitions sont lues, nous obtenons la situation



Les marqueurs "fin de ligne source" ont été remplacés par le code binaire de 127
Les marqueurs "indicateur de paramètre" ont été remplacés par des 0 binaires.

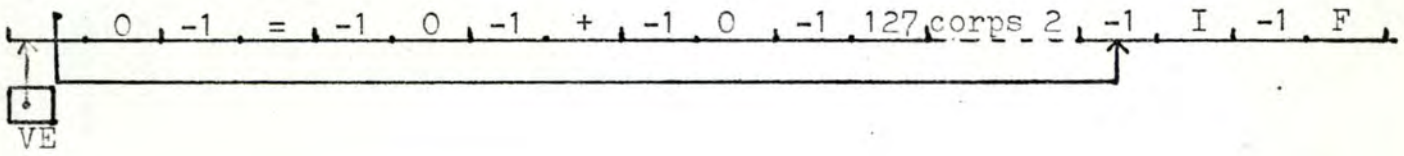
2.- Tri des macrodéfinitions par comparaison des codes internes des caractères dont sont constitués les n clichés lus (code interne = ISOCODE: caractères de 7 bits)

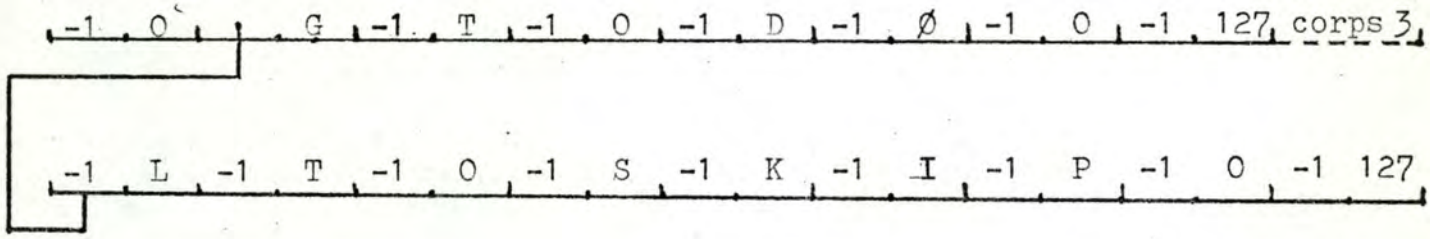
3.- Construction du Template Tree en début de VE.

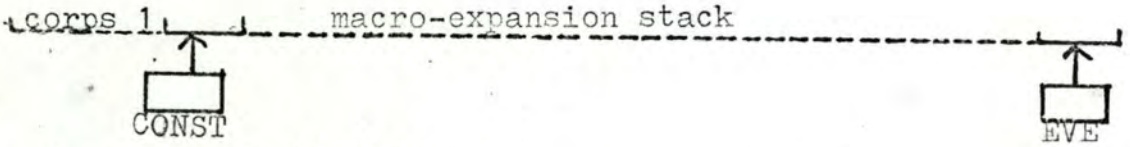
Etant donné les macrodéfinitions

```
IF'LT'SKIP'
corps 1
&
'='+'.
corps 2
&
IF'GT'DO'
corps 3
&&
```

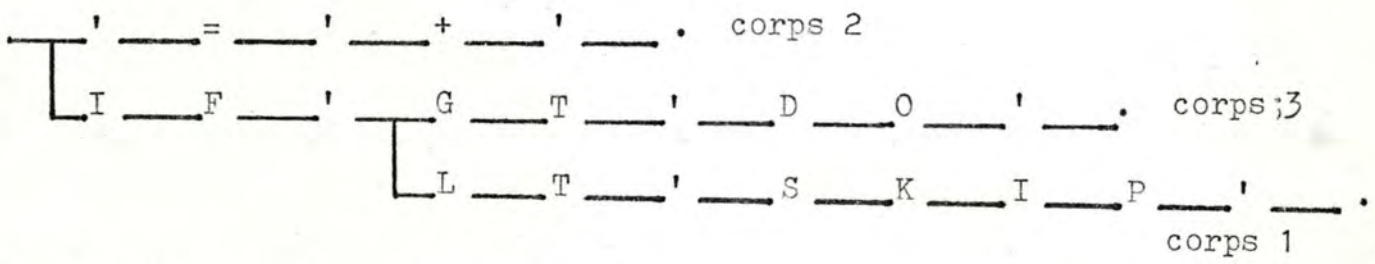
nous obtenons







qui représente l'arbre



Les traits continus dans cette figure représentent les noeuds de l'arbre. A chaque branche (caractère) est associé la direction gauche-droite. Plusieurs branches peuvent partir d'un noeud, mais une seule peut y aboutir.

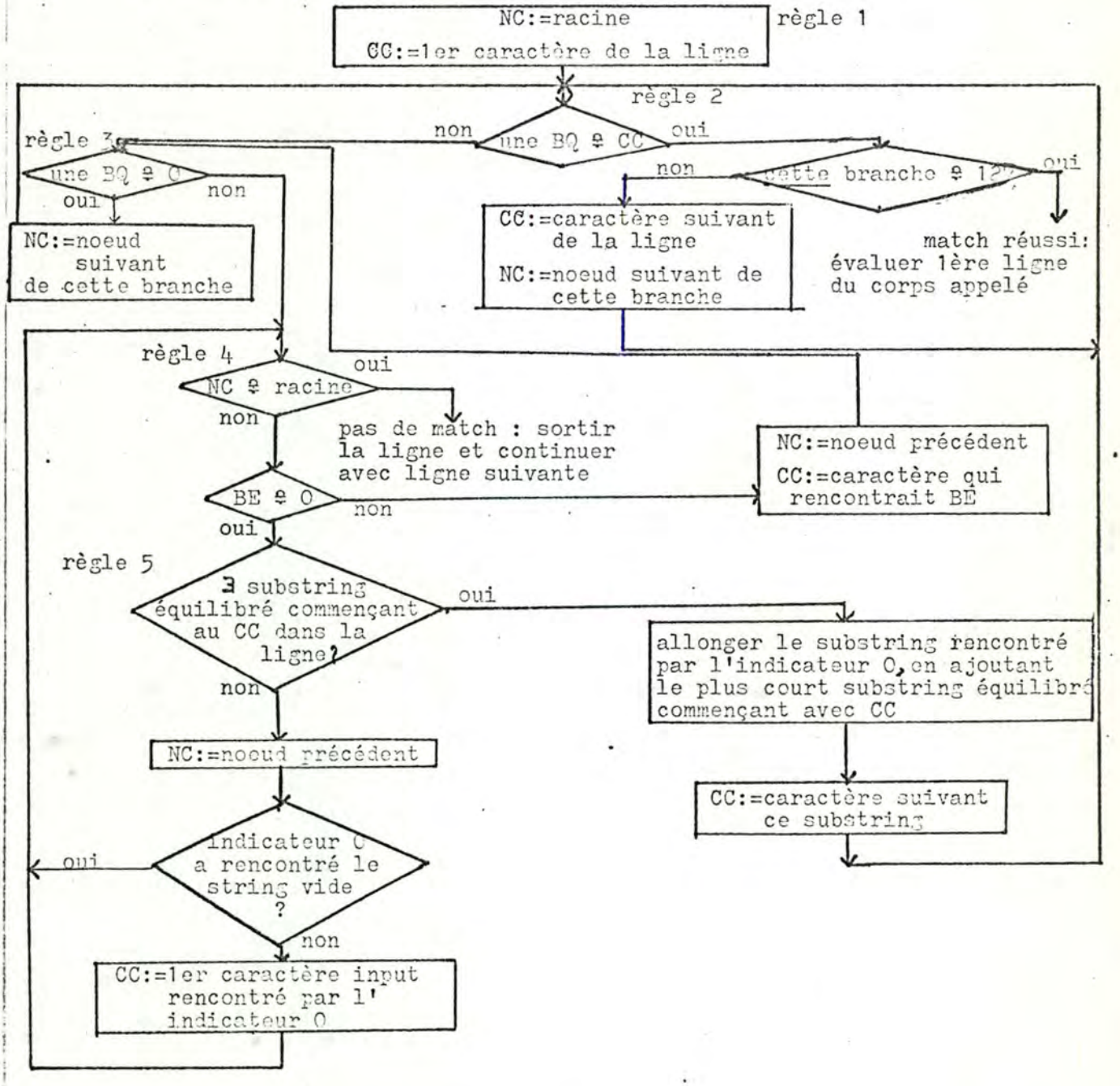
5. Algorithme de "TEMPLATE MATCHING"

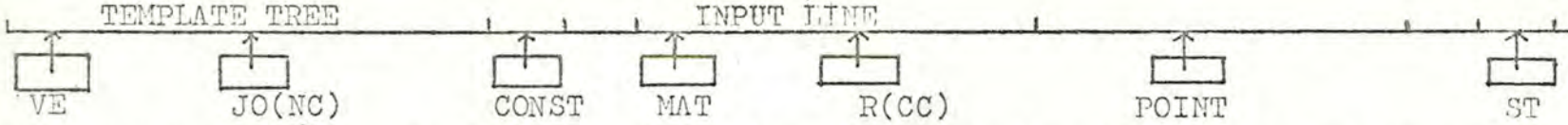
CC caractère courant de la ligne à "matcher"

NC noeud courant

BE branche entrant dans NC

BQ branche quittant NC



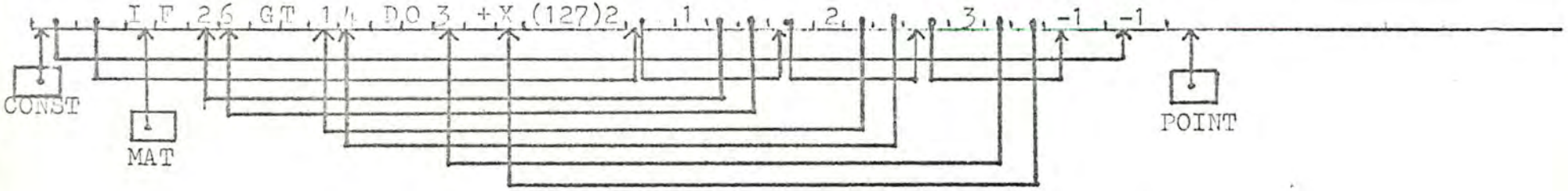


MAT pointe vers le 1er caractère { de l'input line si la ligne appartient au texte de génération
de la "constructed line" si la ligne appartient à un corps

POINT pointe vers le 1er caractère libre du stack d'expansion

JO pointe vers le noeud courant du Template Tree et R vers le caractère courant de la ligne. Chaque fois que !R (! est l'opérateur d'indirection) = !(JO+1), on mémorise l'adresse du noeud courant dans !ST. Les mots du vecteur ST contiennent donc les adresses des noeuds qui précèdent les différentes branches rencontrées. C'est grâce à ce stack ST que des retours en arrière sont possibles (ils sont nécessaires pour tenter de rencontrer un autre cliché lorsque le cliché courant ne convient pas).

Etant donné la ligne input IF26GT14D03+X
nous obtenons en fin de Template Matching :



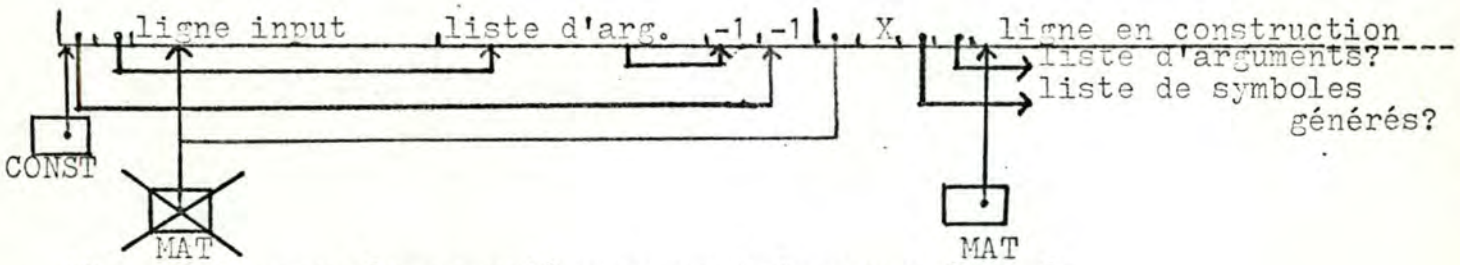
JO pointe à ce moment vers le 1er caractère de "corps 3"

La ligne qui a rencontré un cliché est suivie de deux chaînes, terminées toutes deux par un '-1' binaire :

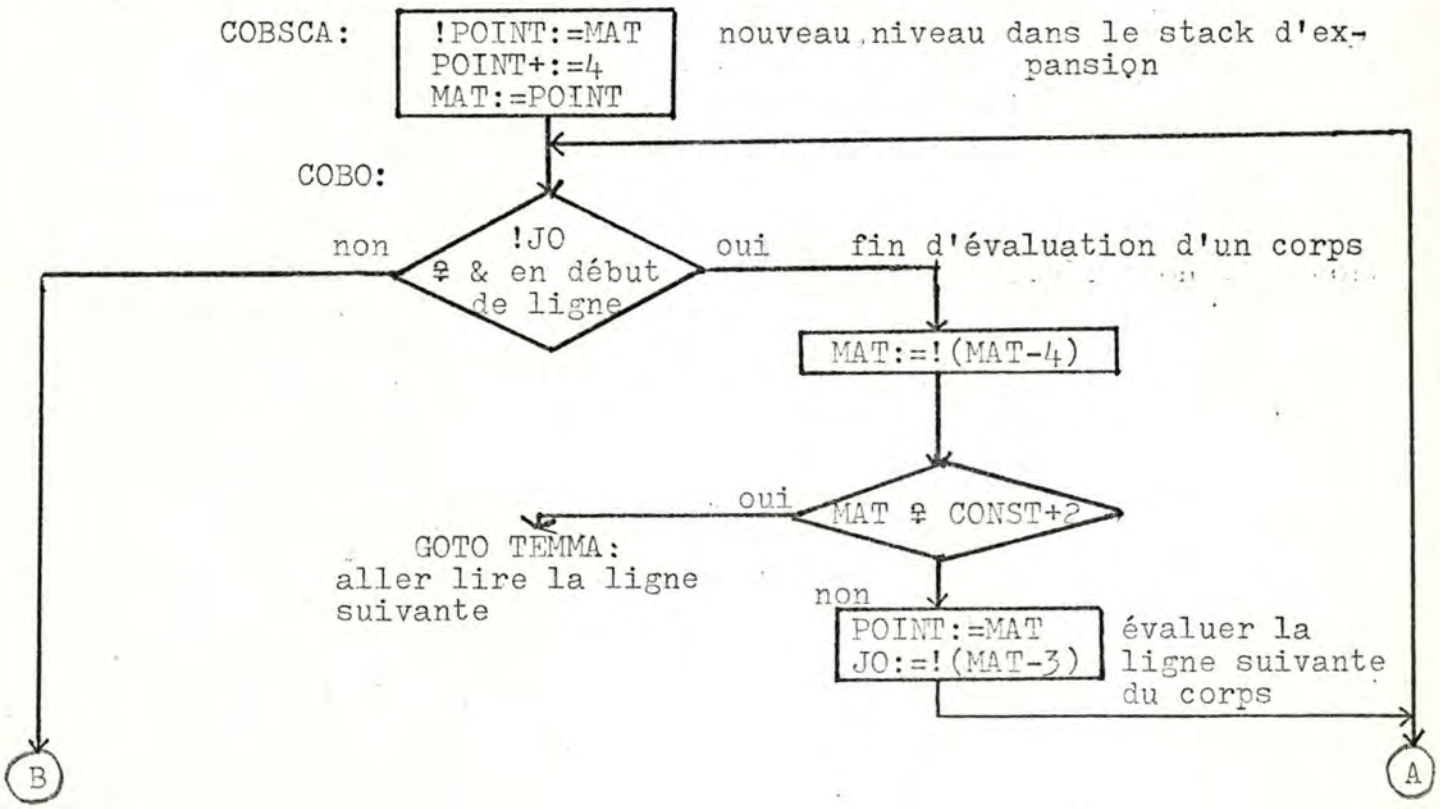
- la chaîne des arguments [pointeur !(MAT-1)]
- la chaîne des symboles générés locaux, vide à ce moment [pointeur !(MAT-2)]

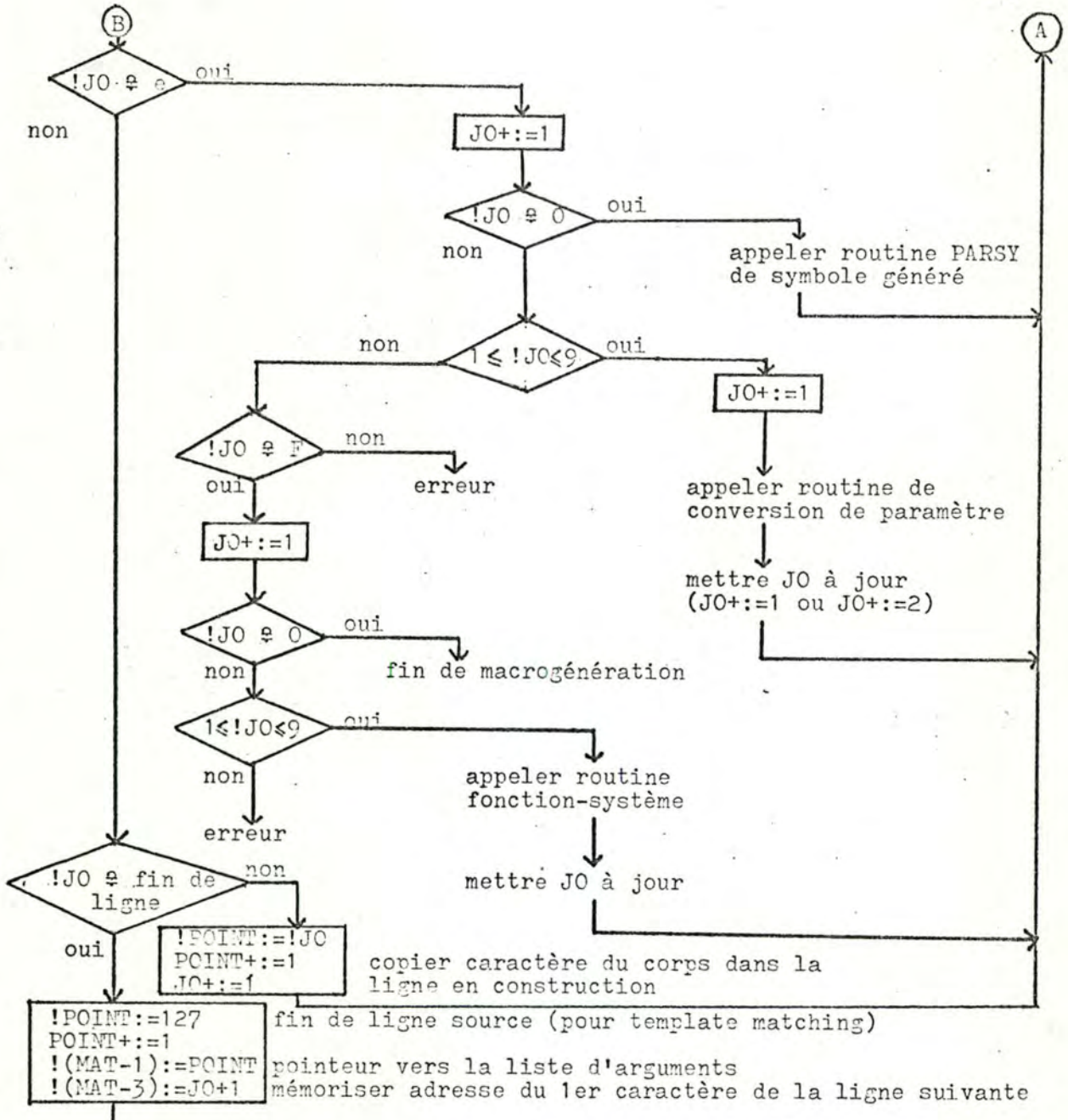
6. Algorithme d'évaluation des corps

Au début de l'évaluation d'une ligne d'un texte de remplacement, il faut ajouter un niveau au stack d'expansion :



JO pointe vers le caractère courant du corps évalué
MAT pointe toujours vers la ligne construite courante
Le mot 'MAT-3' sert à mémoriser l'adresse du caractère suivant, pendant l'expansion des appels imbriqués de niveau supérieur.

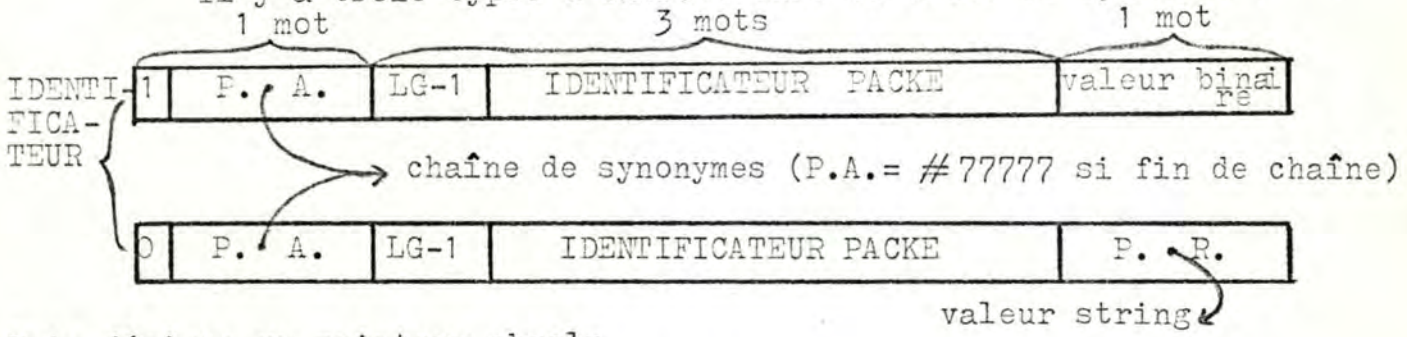



B
!JO ≠ e
oui
non
JO+:=1
!JO ≠ 0
oui
non
appeler routine PARSY
de symbole généré
A
non
1 ≤ !JO≤9
oui
JO+:=1
!JO ≠ F
non
erreur
oui
appeler routine de
conversion de paramètre
JO+:=1
mettre JO à jour
(JO+:=1 ou JO+:=2)
!JO ≠ 0
oui
fin de macrogénération
non
1≤!JO≤9
oui
appeler routine
fonction-système
non
erreur
!JO ≠ fin de
ligne
non
mettre JO à jour
oui
!POINT:=!JO
POINT+:=1
JO+:=1
copier caractère du corps dans la
ligne en construction
!POINT:=127
POINT+:=1
!(MAT-1):=POINT
!(MAT-3):=JO+1
fin de ligne source (pour template matching)
pointeur vers la liste d'arguments
mémoriser adresse du 1er caractère de la ligne suivante
GOTO TEMM pour essayer de "matcher" la ligne construite avec un cliché

7. Table des symboles

Les symboles ont une longueur maximale de 6 caractères et doivent commencer par un caractère alphabétique.

Il y a trois types d'entrées dans la table de symboles :



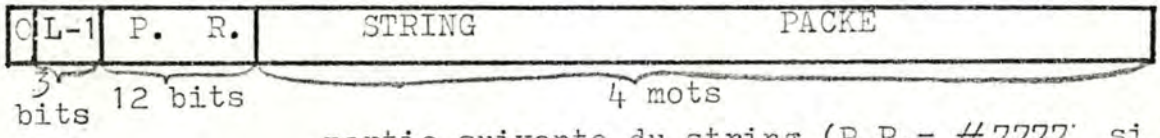
P.A. désigne un pointeur absolu

P.R. désigne un pointeur relatif (déplacement par rapport à SYTAB)

le premier bit du premier mot est

- 1 si le symbole est de type entier
- 0 si le symbole est de type string

La première moitié du premier mot est la "longueur -1" de l'identificateur



partie suivante du string (P.R.= #7777 si dernière partie)

le string est découpé en substrings de 8 caractères

L-1 désigne la "longueur-1" du substring qui se trouve dans cette entrée.

Procédures

PACK : package d'un string (2 caractères par mot)

EXFREE: recherche de la première entrée libre dans la table des symboles (free list)

SRCH : recherche d'un identificateur

VALID : recherche d'une valeur

CRID : création d'un identificateur

ASSID : création d'une valeur

8. Arithmétique entière

DEC : procédure de conversion binaire-décimal
BIN : procédure de conversion décimal-binaire
NUM : test si string numérique, et si oui, conversion de ce string en binaire
ARITH: procédure de calcul d'expressions arithmétiques

9. Différences principales par rapport à STAGE2

- pas d'entrées-sorties généralisées : notre configuration d'I/O est comme suit :
  - un fichier pour le texte source
  - un fichier pour le texte cible
  - un fichier pour les messages d'erreurs

  D'où : 1.- fonction-système eF1 (manuel STAGE2 p. 65):eF1me n'existe pas dans notre version
  2.- fonction-système eF2 (manuel STAGE2 p. 66-67)simplifiée : nous avons gardé uniquement la possibilité de recopier ou d'ignorer des lignes du texte de génération
- restrictions sur l'écriture des symboles (6 caractères maximum, dont le premier doit être alphabétique)
- conversion de paramètre 4 (manuel STAGE2 p. 57) : une variable indéfinie ou à valeur non-numérique dans une expression arithmétique provoque un message d'erreur
- conversion de paramètre 8 (manuel STAGE2 p. 61) : pas de message d'erreur; la valeur décimale du code interne du premier caractère est générée si l'argument est constitué de plusieurs caractères
- fonctions-système 4 à 6 (manuel STAGE2 p. 28) : dans notre version, le compteur de "skips" SK est testé dans les procédures FUN4, FUN5 et FUN6 au lieu de l'être avant chaque lecture ou construction de ligne.
- fonction-système 7 (manuel STAGE2 p. 72) : dans notre version, la portée d'une itération doit toujours être terminée explicitement par eF8.

```
1  GET "LIB3/CHEAD"
2  NEEDS SELECTINPUT.SELECTOUTPUT.RDCH.WRCH.WRITEF
3
4  GLOBAL $(
5  //FLAG LINE
6      SELF:91: SPAF:92: TELF:93: ESC:94: ZERO:95: PADD:96:
7      LPA:97: PLUS:98: MINU:99: MUL:100: DIV:101: RPA:102
8  //GLOBAL VARIABLES
9      SK:103: SYSEN:104: ITOP:105: INFI:106: OUTFI:107
10 //WORKSPACE
11     VE:108: JO:109: CONST:110: MAT:111: POINT:112:
12     PASY:113: CR:114: EVE:115
13 //TABLE
14     HTAB:116: SYTAB:117: TABI:118: FREE:119: IOP:120:
15     CC:121: LT:122
16 //ROUTINES AND FUNSTIONS
17     COMPRS:123: TEMMAT:124: REAPLA:125: READIN:126: WRERR:127:
18     WRILI:128: SCH:129: PARSY:130: PARO:131: PAR12:132:
19     PAR3:133: PAR4:134: ARITH:135: NUM:136: DEC:137: BIN:138:
20     PAR5:139: PAR6:140: PITER:141: PAR8:142: FUN1:143:
21     FUN2:144: FUN3:145: PACK:146: EXFREE:147: SRCHID:148:
22     CRID:149: VALID:150: ASSID:151: FUN4:152: SKIP:153:
23     FUN5:154: FUN6:155: FUN9:156
24   $)
25
26
27
```

```
1  //ALL POINTERS INSIDE VE (STAGE2 WORKING SPACE) ARE ABSOLUTE
2  GET "/VAR"
3  START: S(ST2
4  LET V=VEC 5000
5  LET ROTAB=VEC 2127
6  LET IDPAC=VEC 2
7  INPI:=ZWORD(200)
8  OUTPI:=ZWORD(201)
9  VE.FREE.SYGEN.HTAB:=V.0.1.ROTAB
10 IDP.VE.ILP:=0.@(VE!5000).IDPAC
11 GR.POINT.SYTAB:=EVE+9.VE.HTAB+128
12 REAPLA()
13 FOR I=VE TO EVE DO !I:=#C0141
14 //INITIALISE SYMBOLS TABLE'S FREE LIST AND HASH TABLE
15 FOR I=HTAB TO SYTAB DO !I:=#77777
16 FOR I=0 TO 1991 BY 5 DO SYTAB!I:=I+5
17 SYTAB!1995:=#7777
18
19 //BUILDING OF T T AT FRONT OF VE
20
21 //READING TEMPLATES AT END OF VE
22 S(1B LET PO=VEC 100 //MAX 100 TEMPLATES
23       LET POINTE.POINTD.Y=EVE.PO.0
24       LET COMPRS(J.K)=    //TRUE IF !(!J)<=!(!K)
25           VALOF S( LET X.Y=0.0
26                    J.K:=!J.!K
27                    S( X.Y:=!J.!K : J-:=1 : K-:=1 S) REPEATUNTIL
28                        X\=Y\/X=127\/Y=127
29                    RESULTIS X<=Y S)
30       SELECTINPUT(INPI)
31 NEXTT: !POINTD:=POINTE  //READING A TEMPLATE : CURRENT CELL OF PO
32        POINTD+:=1       //POINTS (ABS) TO FRONT OF TEMPLATE READ
33   S(A Y:=RDCH()          //AT END OF VE
34       !POINTE:=Y
35       IF Y=SELF THEN S( Y:=RDCH() REPEATUNTIL Y="*N" S)
36       IF Y="*N" THEN  //READING A CODE BODY
37       S(1 !POINTE:=127       //SPECIAL MARKER FOR SOURCE EOL FLAG
38           POINTE-:=1
39       S(2 Y:=RDCH()       //NEXT LINE OF CURRENT CODE BODY
40           IF Y=SELF THEN  //END   OF      "     "   "
41              S(3 !POINTE:=SELF
42                  POINTE-:=1
43                  Y:=RDCH()
44                  IF Y=SELF GOTO SORT  //END OF DEFINITIONS
45                  UNTIL Y="*N" DO Y:=RDCH()
46                  GOTO NEXTT S)3
47       S(3 IF Y=ESC THEN   //COPY ESC AND NEXT SYMBOL
48              S( !POINTE:=ESC
49                 !(POINTE-1):=RDCH()
50                 POINTE-:=2
51                 Y:=RDCH()
52                 LOOP S)
53           IF Y=SELF THEN S( Y:=RDCH() REPEATUNTIL Y="*N" S)
54           IF Y="*N" THEN  //END OF A CODE BODY LINE
55              S( !POINTE:="*N"
```

```
                POINTP-:=1
                BREAK  S)
            !POINTP:=Y
            POINTP-:=1
            Y:=RDCH()  S)3 REPEAT S)2 REPEAT S)1
        IF Y=SPAP THEN !POINTP:=0  //SPECIAL MARKER FOR SOURCE PAR FLAG
        POINTP-:=1  S)A REPEAT

//SORTING TEMPLATES (CHANGING POINTERS PO)
SORT: POINTD-:=PO     //POINTD := # TEMPLATES
        Y:=1
        S(X LET N.PAS=0.0  //Y LEAST POWER OF 2 >= POINTD
            Y*:=2<>N+:=1 REPEATUNTIL Y>=POINTD
            PAS:=Y
            WHILE PAS>1 DO
            S(1 PAS/:=2
                S( LET I.LIM=PO-1.PO+POINTD-PAS-1
                   WHILE I<LIM DO
                   S(2 I+:=1
                       S( LET J=I
                          LET K=J+PAS
                          UNTIL J<PO\/COMPRS(J.K) DO
                          S(3 LET X=!J
                              !J:=!K
                              !K:=X
                              J-:=PAS ; K-:=PAS   S)X

//TEMPLATE TREE
S(1  LET I=PO
  S(8 S( !POINT.!(POINT+1):=-1.!(!I)  //TEMPLATE WITH ABS POINTERS
          POINT+:=2                   //FOR NODES
          !I-:=1 S) REPEATUNTIL !(POINT-1)=127
       S( !POINT:=!(!I)               //CODE BODY
          POINT+:=1
          !I-:=1 S) REPEATUNTIL !(POINT-1)=TELE/\!(POINT-2)="*N"
          I+:=1
          IF I=PO+POINTD GOTO TM
       S( LET K=VE-1                  //NEXT TEMPLATE
      AA: K+:=2 ; !I-:=1
     LOP: IF !K=!(!I+1)  GOTO AA
          IF !(K-1) \=-1 THEN S( K:=!(K-1)+1 ; GOTO LOP S)
          !(K-1):=POINT
          !I+:=1  S)B REPEAT    S)TB

TM: CONST:=POINT
    TEMMAT()
    FINISH  S)ST2
.
```

```
GET "/VAR"
//TEMPLATE MATCHING PROCEDURE

LET TEMMAT() BE
//MATCHING OF INPUT LINE
S(TEMCO
TEMMA: POINT:=CONST+2
       MAT:=POINT   //ADDR OF INPUT LINE IN V
       READIN()
       !(CONST+1):=POINT       //POINTER TO ACT PARAM 1
//MATCHING OF CONSTRUCTED LINE     R POINTS TO CURRENT SYMBOL
TEM:
S(IM LET R,NP,PAR=MAT,0,0      //OF INPUT OR CONSTR LINE
     LET ST=POINT+50 //STACK POINTING TO MEETED NODES
     JO:=VE    //POINTS TO CURRENT NODE IN T T
IF ST>CR-150 THEN
   S( LET X=OUTPUT
      SELECTOUTPUT(1)
      WRITES("*NWORKSPACE TOO SMALL : 'NEXT INPUT LINE*N")
      SELECTOUTPUT(X)
      WRERR()
      GOTO TEMMA  S)
//RULE 2
S(1 LET I=JO
   S(2 IF !R=!(JO+1) BREAK
       IF !JO=-1\/!R<!(JO+1) THEN S( JO:=I : GOTO CC1 S)
       JO:=!JO  S)2 REPEAT
   TEST !(JO+1)\=127
   THEN S( !ST:=1
           R+:=1; ST+:=1; JO+:=2
           LOOP S)
//MATCH SUCCEEDS
   OR S( LET U,Q=1,!(MAT-1)
         !POINT:=-1  //END PAR CHAIN  MARKER
         POINT+:=1
         !(MAT-2):=POINT   //ADDR GENERATED SYMBOLS
         !POINT:=-1  //END SYMB GEN CHAIN  MARKER
         POINT+:=1
         UNTIL !Q=-1 DO  //PLACE ACT PARAM NUMBERS
              S( !(Q+1):=ZERO+U
                 Q:=!Q  //OR Q+:=4
                 U+:=1  S)
         JO+:=2
         GOTO COBSCA  S)
//RULE 3
CC1: IF !(JO+1)=0
 S(2 !ST:=JO
     ST+:=1
     JJ+:=2
     NP+:=1     //PARAMETER NUMBER
     PAR:=POINT      //ADDRESS CELL POINTING TO NEXT PARAM
     POINT+:=4
     !PAR:=POINT  //POINTER TO NEXT PARAM
     !(PAR+2):=R  //POINTER TO 1ST CHAR OF PAR
     !(PAR+3):=0  //EMPTY PAR
```

```
      LOOP  S)2
//RULE 4
CC2: IF JO=VE THEN
//MATCH FAILS: OUTPUT LINE
       S(2 WRILI()
           TEST MAT=CONST+2
           THEN GOTO TEMMA
           OR S( JO:=!(MAT-3)
                 POINT:=MAT
                 GOTO COBO  S)2

UNLESS !(JO-1)=0 THEN
    S(2 ST-:=1
        JO:=!ST
        R-:=1
        GOTO CC1    S)2
//RULE 5
IF !R=127 GOTO CC3
//LOOKING FOR THE SHORTEST BALANCED SUBSTRING
TEST !R=LPA
THEN  S(2 LET G.I.C=R.0.1
          S(3 R+:=1
              IF !R=127 S( JO:=VE : GOTO CC2 S)
              TEST !R\=RPA
              THEN S( IF !R=LPA THEN C+:=1
                      LOOP S)
              OR S( C-:=1
                    IF C<0  S( JO:=VE : GOTO CC2 S)
                    IF C=0 THEN   BREAK S)3 REPEAT
           R+:=1  S)2
OR  R+:=1
!(PAR+3):=R-1  //POINTER TO LAST CHAR OF PAR
LOOP
//NO BALANCED SUBSTRING
CC3: ST-:=1
     JO:=!ST
     R:=!(PAR+2)
     NP-:=1
     POINT:=PAR
     PAR-:=4
     GOTO CC2 S)1         REPEAT     S)TM


//CODE BODY SCANNER
COBSCA:
//SCANNING OF NEW CODE BODY AFTER T M
!POINT:=MAT
POINT+:=4
MAT:=POINT

//SCANNING NEXT LINE OF CURRENT CODE BODY
//TEST END OF CURRENT CODE BODY
COBO:
S(1 IF !JO=TELE/\!(JO-1)="*N" THEN
    S( MAT:=!(MAT-4)
```

```
111         TEST MAT=CONST+2
112         THEN GOTO TEMMA
113         OR S( POINT:=MAT
114               JO:=!(MAT-3)  //NEXT SYMBOL TO SCAN IN CODE BODY
115               LOOP S)   S)
116
117  //TEST CURRENT CHARACTER OF CODE BODY
118  //ESCAPE
119  TEST !JO=ESC
120  THEN  S(ES  JO+:=1
121               IF !JO=ESC THEN  //ESC-ESC
122                  S( !POINT:=ESC
123                     POINT+:=1
124                     JO+:=1
125                     LOOP  S)
126               IF !JO=TELF THEN  //ESC-EOL
127                  S( !POINT:=TELF
128                     POINT+:=1
129                     JO+:=1
130                     LOOP  S)
131
132  //PARAMETER CONVERSION
133               IF !JO=ZERO THEN      //SYMBOL GENERATION
134                    S( PARSY()
135                       LOOP  S)
136
137               IF !JO>=ZERO+1/\!JO<=ZERO+9 THEN
138  S(Z JO+:=1
139  //PAR CONC 0
140  IF !JO=ZERO THEN  //EXACT COPY OF ACT PAR  IN CONSTR LINE
141       S( PAR0()
142          JO+:=1 ; LOOP S)
143
144  //PAR CONV 1 2
145  IF !JO=ZERO+1\/!JO=ZERO+2  //COPY VALUE OF PAR TAKEN AS IDENT
146       S( PAR12()             //IN CONSTR LINE
147          JO+:=1 ; LOOP S)
148
149  //PAR CONV 3
150  IF !JO=ZERO+3  //CHAR FOLLOWING PARAM IN INPUT LINE
151      S( PAR3()
152         JO+:=1
153         LOOP  S)
154
155  //PAR CONV 4
156  IF !JO=ZERO+4 THEN  //EVALUATE PARAM AS ARITHM EXPR
157        S( PAR4()
158           JO+:=1
159           LOOP  S)
160
161  //PAR CONV 5
162  IF !JO=ZERO+5     //LENGTH OF PARAM
163      S( PAR5()
164         JO+:=1
165         LOOP  S)
```

```
165
167  //PAR CONV 6
168  IF !JO=ZERO+6   //CONSTR LINE BECOMES ACT PAR
169       S( IF POINT>CR-150 GOTO TEM
170          PAR6()
171          JO+:=2 : LOOP S)
172
173  //PAR CONV 7
174  IF !JO=ZERO+7   //CONTEXT CONTROLLED ITERATION
175     S( IF POINT>CR-150 GOTO TEM
176        PITER()
177        LOOP S)
178
179  //PAR CONV 8
180  IF !JO=ZERO+8  //DECIM REPRES 1ST LETTER OF PARAM
181      S( PAR8()
182         JO+:=1
183         LOOP  S)
184
185  S( LET X=OUTPUT
186     SELECTOUTPUT(1)
187     WRITEF("+NPARAM CONV '%C' NOT PRESENT+N",!JO)
188     SELECTOUTPUT(X)
189     WRERR()
190     JO+:=1
191     LOOP   S)2
192
193  //PROCESSOR FUNCTION
194                  IF !JO='F' THEN
195  S(2 JO+:=1
196
197  //FUNCTION 0
198  IF !JO=ZERO THEN RETURN  //END OF MACROGENERATION
199
200  //FUNCTION 1
201  IF !JO=ZERO+1 THEN   //NO MATCHING ESSAY OR NEXT LINE = TEMPL FIELD
202       S( FUN1(): LOOP S)
203
204  //FUNCTION 2
205  IF !JO=ZERO+2 //COPY INPUT TEXT
206       S( JO+:=1
207          FUN2()
208          GOTO TEMMA S)
209
210  //FUNCTION 3
211  IF !JO=ZERO+3  //ENTRY IN MEMORY
212       S( FUN3()
213          JO+:=2 : POINT:=MAT : LOOP S)
214
215  //FUNCTION 4
216  IF !JO=ZERO+4  //SKIP UNCOND
217     S( FUN4()
218        IF MAT=CONST+2 GOTO TEMMA
219        LOOP S)
220
```

```
221  //FUNCTION 5
222  IF !JO=ZERO+5  //SKIP ON STRING EQUALITY
223      S( JO+:=1
224         FUN5()
225         IF MAT=CONST+2 GOTO TEMMA
226         LOOP S)
227
228  //FUNCTION 6
229  IF !JO=ZERO+6  //SKIP ON RELAT VALUE OF 2 A.E.
230     S( JO+:=1
231        FUN6()
232        IF MAT=CONST+2 GOTO TEMMA
233        LOOP  S)
234
235  //FUNCTION 7
236  IF !JO=ZERO+7      //COUNT CONTROLLED ITERATION
237      S( PITER()
238         LOOP S)
239
240  //FUNCTION 8
241  IF !JO=ZERO+8  //END OF ITERATION?
242      S( PITER()
243         LOOP S)
244
245  //FUNCTION 9
246  IF !JO=ZERO+9   //SKIP REMAINDER OF CURRENT CODE BODY
247     S( JO+:=2
248        FUN9()
249        LOOP S)
250
251  S( LET X=OUTPUT
252     SELECTOUTPUT(1)
253     WRITEF("+NFUNCT '%C' NOT PRESENT+N",!JO)
254     SELECTOUTPUT(X)
255     WRERR()
256     JO+:=1
257     LOOP  S)2
258
259  //UNUSUAL CHAR AFTER ESC : COPY IT IN CONSTR LINE
260  S( LET X=OUTPUT
261     SELECTOUTPUT(1)
262     WRITEF("+NUNUSUAL CHAR '%C' AFTER ESC+N",!JO)
263     SELECTOUTPUT(X)
264     WRERR()
265     JO+:=1
266     LOOP     S)ES
267
268  //ONE LINE OF CODE BODY CONSTRUCTED : TRY MATCH
269  OR S( TEST !JO="+N"
270        THEN S( !POINT:=127
271                POINT+:=1
272                !(MAT-1):=POINT  //POINTER TO ACT PAR 1
273                !(MAT-3):=JO+1  //NEXT SYMBOL TO SCAN IN CODE BODY
274                GOTO TEM  S)
275        OR S( !POINT:=!JO     //COPY CHAR OF CODE BODY IN CONSTR LINE
```

```
276             POINT←:=1 : J0←:=1  $)1  REPEAT     $ITEMCO
277
278
279
280
281
282
283
```

```
GET "/VAR"

LET PACK(PD,PF,J) BE     //PACKS IDENT . EXCEPT 1ST CHAR REPLACED BY
    $( LET I=PD          //LENGHTH-1 . IN CELLS J TO J+2
       !PD:=PF-PD     //LENGTH-1 IN 1ST HALFWORD
       WHILE I<PF DO $( !J:=!I<<8\/!(I+1) : J+:=1 : I+:=2 $)
       IF I=PF THEN !J:=!I<<8
       !PD:=CC  //RESTORE 1ST CHAR
       RETURN  $)

AND EXFREE()=  //GIVES REL ADDR NEXT FREE ENTRY OF SYTAB AND TAKES
    VALOF $( LET X=FREE   //THAT ENTRY AWAY FROM FREE LIST
             IF FREE=#7777 THEN
               $( LET X=OUTPUT
                  SELECTOUTPUT(1)
                  WRITEF("*NSYMBOL TABLE TOO SHORT*N")
                  SELECTOUTPUT(X)
                  WRERR()
                  RESULTIS X $)
             FREE:=SYTAB!FREE
             RESULTIS X $)

//SEARCH IDENT
AND SRCHID(PD,PF)=  //TRUE AND TABI POINTS TO IDENT ENTRY IN SYTAB IF
//ID EXISTS : FALSE AND TO HTAB IF NO SYN.OR TO ENTRY OF LAST SYN
//CHAIN IN SYTAB. IF ID DOES NOT EXIST
  VALOF $( IF PF-PD>5 THEN
             $( LET X=OUTPUT
                SELECTOUTPUT(1)
                WRITEF("*NIDENTIF TOO LONG*N")
                SELECTOUTPUT(X)
                WRERR()
                TABI:=0
                RESULTIS FALSE $)
           CC:=!PD //SAVE 1ST CHAR OF IDENT: NEXT LINE HASH FUNCTION
           TABI:=((CC/\31)\/((!PF/\3)<<5))+HTAB  //TABI TO HTAB
           IF !TABI=#77777 THEN RESULTIS FALSE //NO IDENT STARTING
           TABI:=!TABI //TABI TO ID ENTRY IN SYTAB //WITH SAME CHAR
           PACK(PD,PF,IDP)
           $( FOR L=0 TO (PF-PD)/2 DO  //COMPARE
                $( IF IDP!L\=!(TABI+1+L) GOTO SYN  $)
           RESULTIS TRUE
       SYN: IF !TABI<<1>>1=#77777 THEN RESULTIS FALSE //NOT IN SYN
            TABI:=!TABI/\#77777 $) REPEAT $) //ADDR NEXT SYN

//SEARCH VALUE
AND VALID(A) = //TRUE IF NUM.FALSE IF ALPHAB: VALUE PUT IN A....
  VALOF $( LET PT=0 //TABI POINTS TO IDENT ENTRY  //LENGTH IN LT
           IF !TABI>>15=0 GOTO ALPHA
           !A:=!(TABI-4)  //IF NUM. VALUE WAS IN 5TH WORD OF ID ENTRY
           RESULTIS TRUE
    ALPHA: PT:=@(SYTAB!(!(TABI+4)))//REL POINTER TO VALUE ENTRY WAS
           LT:=0                   // 5TH WORD OF ID ENTRY
         $( LET L,B=!PT>>12,A-2 //LENGTH-1 OF VALUE WAS IN BITS
           A-:=1       //2 TO 4 OF 1ST WORD OF VALUE ENTRY
```

```
            FOR I=1 TO 4 DO  //VALUE WAS IN WORDS 2 TO 5 OF VAL ENTRY
              S( B!(I+2):=PT!I>>R   //UNPACK VALUE AND
                 A!(I+2):=PT!I/\#377 S) //EVENTUAL REST. COPY IT
            PT:=(!PT<<4)>>4  //REL POINTER TO NEXT VALUE PART WAS IN
            LT+:=L+1         //1ST WORD OF VAL ENTRY
            IF PT=#7777 THEN   RESULTIS FALSE        //END OF VALUE
            PT+:=SYTAB: A+:=9 S)  REPEAT S)

//CREATE IDENTIFICATOR
AND CRID(PD,PF) BE //TABI POINTS TO LAST ENTRY IN SYN CHAIN OR TO HTA
    S( LET Y=0
       Y:=@(SYTAB!EXFREE()) //Y ABS POINTER TO NEXT FREE ENTRY
       TEST TABI<SYTAB //IF TRUE,TABI POINTED TO HTAB
       THEN !TABI:=Y  //CREATE ABS POINTER TO SYTAB IN HTAB
       OR S( !TABI:=(!TABI/\#100000)\/Y  S)  //SAVE TYPE BIT OF
          //LAST SYN AND PUT ABS POINTER TO NEXT SYN IN 1ST WORD
       TABI:=Y          //TABI TO ID ENTRY
       !TABI:=#177777  //INITIALIZE 1ST WORD OF IDENT ENTRY
       PACK(PD,PF,TABI+1) //PACK IDENT IN 2D TO 4TH WORDS OF ID ENTRY
       RETURN S)

//CREATE VALUE
AND ASSID(PD,PF,TY) BE   //PUT VALUE IN TABLE
//TABI POINTS TO ID ENTRY: TYPE TRUE IF NUM,FALSE IF ALPHA
    S( LET Y=FREE //1ST BIT OF !TABI = 1 IF NUM VALUE OR IF ID JUST
       IF !TABI>>15=0          //CREATED
      S(1 LET Z,Z1=0,0 //BEFORE REPLACING OLD ALPHAB VALUE BY NEW.
          FREE:=!(TABI+4) //UPDATE FREE LIST : THE CHAIN OF VALUES
          Z:=FREE         //IS RETURNED TO FREE LIST
          S( Z1:=Z : Z:=(SYTAB!Z/\#7777) : SYTAB!Z1:=Z  S)
                REPEATUNTIL Z=#7777
          SYTAB!Z1:=Y S)1
       IF \TY GOTO ALPH
       !TABI\/:=#100000  //NUM. THUS 1ST BIT OF ID ENTRY PUT TO 1
       !(TABI+4):=!PD    //UPDATE NUM VALUE
       RETURN
  ALPH: Y:=EXFREE()  //UPDATE ALPH VALUE
        !(TABI+4):=Y  //PUT REL POINTER TO VAL ENTRY IN 5TH WORD IDEN
        !TABI/\:=#77777  //1ST BIT OF ID ENTRY TO 0
       S( LET Y1=0
          Y+:=SYTAB-1  //ABS POINTER TO VALUE ENTRY
          FOR I=0 TO 3 DO  //PACK VAL AND REST IN2D TO 5TH WORD
             S( Y1!:=!PD<<8\/PD!1 : PD+:=2 S)
          IF PD>PF BREAK
          Y1:=EXFREE()
          !(Y-1):=Y1\/#70000  //REL POINTER TO NEXT VALUE PART AND
          Y:=Y1  S) REPEAT  //LENGTH-1 IN 1ST WORD OF VAL ENTRY
          !(Y-1):=#7777\/((8-PF-PD)<<12)  //LFGTH-1 OF LAST VAL PART
          RETURN S)      //AND "BIDON" POINTER

AND FUN3() BE  //PAR1->IDENT,PAR2->VALUE
  S( LET X=0
     IF \SCH(ZERO+1,!(MAT-4)-1) GOTO XY  //INEXIST PARAM
     IF !(PASY-3)=0 GOTO XY  //EMPTY PARAM
     X:=!(!(PASY-2))
```

```
111       IF X<'A'\/X>'Z' GOTO XY  //IDENT STARTING WITH DIGIT!
112       X:=PASY
113       IF \SCH(ZERO+2,!(MAT-4)-1) GOTO XY  //INEXIST PARAM
114       IF \SRCHID(!(X+2),!(X+3)) THEN  //NEW IDENT
115            S( TEST TABI=0
116               THEN RETURN  //IDENT TOO LONG
117               OR CRID(!(X+2),!(X+3))  S)
118       IF !(PASY+3)=0  THEN S( !POINT:=#00142   //EMPTY VALUE
119                               ASSID(POINT,POINT,FALSE)
120                               RETURN  S)
121       TEST NUM(!(PASY+2),!(PASY+3))
122       THEN ASSID(POINT,POINT,TRUE)  //PAR2 = NUM STRING
123       OR ASSID(!(PASY+2),!(PASY+3),FALSE) //PAR2 = ALPHA STRING
124       RETURN
125  XY=  S( LET X=OUTPUT
126          SELECTOUTPUT(1)
127          WRITEF("*NFUN3.PAR1 EMPTY OR INEXIST PAR OR PAR1 *
128                 *STARTING WITH DIGIT*N")
129          SELECTOUTPUT(X)
130          WRERR()
131          RETURN  S)   S)
132
133  AND PAR12() BE  //COPY VALUE OF PAR=IDENT
134    S( IF \SCH(!(JO-1),!(MAT-4)-1) GOTO CC5 //INEXIST PARAM
135       IF !(PASY+3)=0 GOTO CC5  //EMPTY PAR, NO IDENT
136       IF !(!(PASY+2))<'A'\/!(!(PASY+2))>'Z' GOTO CC5 //BAD IDENT
137       TEST SRCHID(!(PASY+2),!(PASY+3))
138       THEN S( TEST VALID(POINT)  //COPY VALUE
139               THEN S( LET X=!POINT  //CONVERT TO DEC
140                       DEC(X)  S)
141               OR S( IF !POINT=#00142 RETURN   //EMPTY VALUE
142                     POINT+:=LT  S)  S)       //STRING VALUE
143       OR S(1 IF TABI=0 RETURN  //IDENT TOO LONG
144              TEST !JO=ZERO+1
145              THEN RETURN  //CONV1, UNDEF IDENT, COPY NOTHING
146              OR S( CRID(!(PASY+2),!(PASY+3))  //NEW IDENT
147                    ASSID(@SYGEN,@SYGEN,TRUE)  //VALUE FROM SYGEN
148                    DEC(SYGEN)  //COPY DEC VALUE (IN CONSTR LINE)
149                    SYGEN+:=1 S)1
150        RETURN
151  CC5:  S( LET X=OUTPUT
152           SELECTOUTPUT(1)
153           WRITEF("*NCONV12. PAR EMPTY OR INEXIST OR STARTING DIGIT*N")
154           SELECTOUTPUT(X)
155           WRERR()
156           RETURN  S)   S)
157
158
159
160
161
```

```
1  GET "/VAR"
2
3  LET ARITH(DB,FN) = //IF A.E. CORRECT, TRUE AND BINARY VALUE IN
4    VALOF $(AR LET ST.SO.CNT=POINT+50,0,0       //!POINT
5      IF ST REM 2 = 1  ST+:=1
6      SO:=ST       //SO EVEN
7      FOR I=SO TO SO+100 DO  !I:=0   //INITIALIZE STACK
8  IF !DB>='A'/\!DB<='Z'\/!DB>=ZERO/\!DB<=ZERO+9
9        $( ST+:=1 : GOTO VAR $)   //A.E. STARTS WITH VARIABLE OR LITT
10     DB-:=1
11 $(1 DB+:=1
12 IF !DB=MUL\/!DB=DIV THEN   //* OR /
13     $(2 IF ST REM 2 = 1 $( !SO:='1' : BREAK $)
14         IF ST=SO $( !SO:='2' : BREAK $)
15         IF !(ST-2)=MINU $( !(ST-2):=PLUS
16                            IF !DB=DIV THEN
17                   $( IF !(ST-1) REM 2 = 1 THEN  !(ST-1)-:=1  $)
18                            !(ST-1):=-!(ST-1)
19                            !ST:=!DB
20                            ST+:=1
21                            LOOP $)
22         IF !(ST-2)=LPA\/!(ST-2)=PLUS $( !ST:=!DB
23                                         ST+:=1
24                                         LOOP $)
25         IF !(ST-2)=MUL  $( !(ST-3)+:=!(ST-1)
26                            !(ST-2):=!DB
27                            ST-:=1
28                            LOOP $)
29         !(ST-3)/:=!(ST-1)
30         !(ST-2):=!DB
31         ST-:=1
32         LOOP   $)2
33 IF !DB=PLUS\/!DB=MINU THEN  //+ OR -
34     $(2 IF ST REM 2 = 1 THEN
35                $(3 TEST !(ST-1)=LPA
36                    THEN $( !ST:=0
37                            !(ST+1):=!DB
38                            ST+:=2
39                            LOOP  $)
40                    OR $( !SO:='1' : BREAK $)3
41 REP:     IF ST=SO+2 THEN
42              $(3 IF !SO=MINU  $( !SO:=PLUS
43                                  !(SO+1):=-!(SO+1)  $)
44                  !ST:=!DB
45                  ST+:=1
46                  LOOP  $)3
47         IF ST=SO\/!(ST-2)=LPA $( !ST:=!DB
48                                  ST+:=1
49                                  LOOP $)
50         IF !(ST-2)=PLUS $( !(ST-3)+:=!(ST-1)
51                            !(ST-2):=!DB
52                            ST-:=1
53                            LOOP $)
54         IF !(ST-2)=MINU $( !(ST-3)-:=!(ST-1)
55                            !(ST-2):=!DB
```

```
                                ST-:=1
                                LOOP S)
          IF !(ST-2)=MUL S( !(ST-3)*:=!(ST-1)
                                ST-:=2
                                GOTO REP S)
          !(ST-3)/:=!(ST-1)
          S1-:=2
          GOTO REP  S)2
IF !DB=LPA THEN   //(
    S(2 TEST ST REM 2 = 0
        THEN S(3 TEST ST=S0
                 THEN S( CNT+:=1
                         !S0:=LPA
                         ST+:=1
                         LOOP S)
                 OR S( !S0:='3' : BREAK S)3
        OR S( CNT+:=1
              !(ST+1):=LPA
              ST+:=2
              LOOP S)2
IF !DB=RPA THEN  //)
    S(2 IF ST REM 2 = 1 S( !S0:='3' : BREAK S)
        CNT-:=1
        IF CNT<0 S( !S0:='4' : BREAK S)
        UNTIL !(ST-2)=LPA DO
            S(3 ST-:=2
                IF !ST=PLUS S( !(ST-1)+:=!(ST+1) : LOOP S)
                IF !ST=MINU S( !(ST-1)-:=!(ST+1) : LOOP S)
                IF !ST=MUL  S( !(ST-1)*:=!(ST+1) : LOOP S)
                !(ST-1)/:=!(ST+1) S)3
        IF ST\=S0+2 S( !(ST-3):=!(ST-1)
                       ST-:=2 S)
        LOOP S)2
VAR: IF ST REM 2 =0 S( !S0:='3' : BREAK S)
    S(2 LET X=DB
        UNTIL (!DB=PLUS\/!DB=MINU\/!DB=MUL\/!DB=DIV\/!DB=LPA\/
               !DB=RPA\/DB=FN+1) DO DB+:=1
        DB-:=1
        IF !X>='A'/\!X<='Z' THEN
            S(3 TEST \SRCHID(X,DB)   //VARIABLE
                THEN S( !S0:='5' : BREAK S)  //NON-EXISTENT
                OR S( TEST VALID(POINT)
                      THEN S( !ST:=!POINT
                              ST+:=1
                              LOOP S)
                      OR S( !S0:='5' : BREAK S)3  //STRING VALUE!
        TEST NUM(X,DB)
        THEN S( !ST:=!POINT  //LITTERAL
                ST+:=1
                LOOP S)
        OR S( !S0:='5' : BREAK   S)1  REPEATWHILE DB\=FN
IF !S0>='1'/\!S0<='5' GOTO ERR
IF CNT\=0 S( !S0:='4' : GOTO ERR S)
IF ST REM 2 = 1 S( !S0:='2' : GOTO ERR S)
UNTIL ST=S0+2 DO
```

```
111     S(1 ST-:=2
112         IF !ST=PLUS S( !(ST-1)+:=!(ST+1) : LOOP S)
113         IF !ST=MINU S( !(ST-1)-:=!(ST+1) : LOOP S)
114         IF !ST=MUL  S( !(ST-1)*:=!(ST+1) : LOOP S)
115         !(ST-1)/:=!(ST+1)  S)1
116 IF !S1=MINU THEN !(SO-1):=-!(SO-1)
117 !POINT:=!(SO-1)  //RESULT IN !POINT
118 RESULTIS TRUE
119 ERR: S(1 LET X=OUTPUT
120           SELECTOUTPUT(1)
121           SWITCHON !SO INTO
122 S(S CASE '1': WRITEF("*N2 SUCCESSIVE OPERATORS IN A.E.*N") : ENDCASE
123    CASE '2': WRITEF("*NBAD STARTING OR ENDING A.E.*N") : ENDCASE
124    CASE '3': WRITEF("*NMISPLACED BRACKET OR BAD STARTING A.E.*N")
125              ENDCASE
126    CASE '4': WRITEF("*NUNBALANCED BRACKETS IN A.E.*N") : ENDCASE
127    CASE '5': WRITEF("*NA.E.: IDENT UNDEF OR BAD FORMAT*N") S)S
128           SELECTOUTPUT(X)
129           WRERR()
130           RESULTIS FALSE  S)AR
131 .
132
133
134
135
```

```
Get "//VAR"

LET DEC(A) BE //PUTS DEC FORM OF BINARY NUMBER A IN CELLS POINT...
  $( LET K=1
    IF A<0 DO $( A:=-A
                 !POINT:=MINU
                 POINT+:=1 $)
    UNTIL 10*K>A DO K*:=10
    $( !POINT:=ZERO+A/K
       POINT+:=1
       A REM:=K
       K/:=10  $) REPEATUNTIL K<1
    RETURN $)

AND BIN(A,B) =
  VALOF $( LET I,K=0,1  //CONVERTS NUMBER STRING A-B TO BINARY
           UNTIL B<A DO
              $( I+:=(!B-ZERO)*K
                 K*:=10 : B-:=1 $)
           RESULTIS I $)

AND NUM(A,B) =  //TRUE AND BINARY VALUE IN !POINT IF DEC STRING
   VALOF $( LET X=A
            IF !A=MINU  THEN $( TEST B=A
                                THEN RESULTIS FALSE
                                OR A+:=1 $)
            WHILE A<=B DO
            $( IF !A<ZERO\/!A>ZERO+9 RESULTIS FALSE
               A+:=1  $)
            TEST !X=MINU
            THEN !POINT:=-BIN(X+1,B)
            OR !POINT:=BIN(X,B)
            RESULTIS TRUE  $)

AND SKIP() =   //TRUE AND SKIP EXECUTED IF SK>0
  VALOF $( IF SK<=0 RESULTIS FALSE
           ITOP:=0
           UNTIL SK=0 DO
            $( WHILE !JO=TELE DO $( MAT:=!(MAT-4)
                                    JO:=!(MAT-3)  $)
               IF MAT=CONST+2  THEN
                     $( UNTIL SK=0 DO
                           $( POINT:=MAT : READIN() : SK-:=1 $)
                        RESULTIS TRUE $)
               IF !JO=ESC/\!(JO+1)='F'/\!(JO+2)=ZERO+7 THEN
                    $( ITOP+:=1 : GOTO WE $)
               IF !JO=ESC/\!(JO+1)='F'/\!(JO+2)=ZERO+8 THEN
                    $( IF ITOP>0 THEN $( ITOP-:=1 : GOTO WE $)
                       TEST !CP=0
                       THEN $( CR+:=9 : GOTO WE $) //END COUNT ITER
                       OR $( SCH(!(CR-4),!(MAT-4)-1)//END CONTEXT
                             IF !(CR-3)=TRUE THEN //CONTR ITER
                                  $( !(PASY+2):=!(CR-5)
                                     !(PASY+3):=!(CR-6) $)
                             !(!(MAT-4)-1):=!(CR-8)
```

```
                                MAT:=!CR
                                POINT:=MAT  //??
                                CR+:=9  $) $)
            WE: JO+:=1 REPEATUNTIL !(JO-1)="*N"
                SK-:=1   $)
                RESULTIS TRUE  $)

AND PAR4() BE  //ARITHM EXPR
  $( IF \SCH(!(JO-1),!(MAT-4)-1) THEN
          $( LET X=OUTPUT
             SELECTOUTPUT(1)
             WRITEF("*NCONV4. INEXIST OR EMPTY PARAM*N")
             SELECTOUTPUT(X)
             WRERR()
             RETURN  $)
     IF ARITH(!(PASY+2),!(PASY+3)) THEN
           $( LET X=!POINT
              DEC(X)  $)
     RETURN  $)

AND FUN2() BE //COPY OR DELETE INPUT TEXT
  $(1 IF \SCH(49,!(MAT-4)-1) THEN
          $( LET X=OUTPUT
             SELECTOUTPUT(1)
             WRITEF("*NFUN2. NON-EXIST PARAM*N")
             SELECTOUTPUT(X)
             WRERR()
             RETURN $)
      IF !(PASY+3)=0 RETURN //IF EMPTY PARAM. READ NEXT INPUT LINE
      $( POINT:=MAT
         READIN()
         POINT:=MAT
         FOR I=!(PASY+2) TO !(PASY+3) DO
              $( IF !I\=!POINT GOTO NEXT : POINT+:=1 $)
         RETURN  //DELETE LINE STARTING WITH PAR1 STRING
       //IF !JO="1", COPY INPUT LINE : OTHERWISE DELETE IT
   NEXT: IF !JO=ZERO+1  WRILI() $) REPEAT  $)1

AND FUN9() BE  //SKIP REST OF CODE BODY
  $(1 ITOP:=0
      UNTIL !JO=TELE DO
      $(2 IF !JO=ESC/\!(JO+1)="E" THEN
          $(3 IF !(JO+2)=ZERO+7  $( ITOP+:=1 : GOTO TAL $)
              IF !(JO+2)=ZERO+8 THEN
                   $(4 IF ITOP>0  $( ITOP-:=1 : GOTO TAL $)
                       TEST !CR=0
                       THEN $( CR-:=9 : GOTO TAL $)//END COUNT ITER
                       OR $( SCH(!(CR-4),!(MAT-4)-1)//END CONT ITER
                             IF !(CR-3)=TRUE THEN
                                  $( !(PASY+2):=!(CR-5)
                                     !(PASY+3):=!(CR-6) $)
                             !(!(MAT-4)-1):=!(CR-8)
                             MAT:=!CR
                             POINT:=MAT
                             CR+:=9 $)3
```

```
111             TAL: JO:=1 REPEATUNTIL !(JO-1)="*N" S)1
112 .
113
114
115
116
117
```

```
GET "//VAR"

LET SCH(A,B) = //SEARCH ACT PAR OR GEN SYMB. TRUE IF EXISTS
  VALOF $( B:=!B
           $(1 TEST !B=-1  //B POINTER OF PAR OR GSYMB STRING
               THEN RESULTIS FALSE
               OR $( TEST !(B+1)=A  //A NUMBER OF PAR OR GSYMB
                     THEN $( PASY:=B //PASY POINTS TO PAR OR GSYMB
                             RESULTIS TRUE $)
                     OR B:=!B $)1 REPEAT $)

AND PAR0() BE //COPY ACT PAR IN CONSTR LINE
  $( TEST \SCH(!(JO-1),!(MAT-4)-1)
     THEN $( LET X=OUTPUT
             SELECTOUTPUT(1)
             WRITEF ("*NINEXIST PARAM.CONV 0*N")
             SELECTOUTPUT(X)
             WRERR()
             RETURN $)
     OR $( IF !(PASY+3)=0 RETURN  //EMPTY PARAM
           $( LET I=!(PASY+2)
              WHILE I<=!(PASY+3) DO $( !POINT:=!I
                                       POINT+:=1 : I+:=1 $) $)
              RETURN $) $)

AND PAR3() BE  //CHAR AFTER PARAM IN INPUT STRING
  $( IF \SCH(!(JO-1),!(MAT-4)-1) THEN
             $( LET X=OUTPUT
                SELECTOUTPUT(1)
                WRITEF("*NCONV3. NON-EXIST PARAM*N")
                SELECTOUTPUT(X)
                WRERR()
                RETURN  $)
     IF !(PASY+3)=0 THEN  //EMPTY PARAM
        $( TEST !(!(PASY+2))=127
           THEN RETURN  //AT END OF LINE
           OR $( !POINT:=!(!(PASY+2))
                 POINT+:=1
                 RETURN  $)  $)
     IF !(!(PASY+3)+1)=127  RETURN  //AT END OF LINE
     !POINT:=!(!(PASY+3)+1)
     POINT+:=1
     RETURN  $)

AND PAR5() BE  //LENGTH OF PARAM
    $( IF \SCH(!(JO-1),!(MAT-4)-1) THEN
           $( LET X=OUTPUT
              SELECTOUTPUT(1)
              WRITEF("*NCONV5. NON-EXIST PARAM*N")
              SELECTOUTPUT(X)
              WRERR()
              RETURN  $)
       IF !(PASY+3)=0   $( !POINT:=ZERO
                           POINT+:=1
                           RETURN  $)
```

```
56          $( LET X=!(PASY+3)-!(PASY+2)+1
57             DEC(X)
58             RETURN  $)  $)
59
60  AND PAR6() BE //CONSTR LINE BECOMES ACT PAR  .
61      $(1 LET Z=!(MAT-4)-1
62          IF \SCH(!(JO-1),Z) THEN //ADD TO ACT PAR STRING
63           $( !(MAT-1):= POINT=MAT ->0,POINT-1 //2D POINTER
64              !(MAT-2):=MAT  //1ST POINTER
65              !(MAT-3):=!(JO-1)
66              !(MAT-4):=!Z
67              !Z:=MAT-4
68              GOTO DD $)
69          !(PASY+2):=MAT  //UPDATE ACT PAR STRING
70          !(PASY+3):= POINT=MAT ->0,POINT-1
71      DD: !POINT:=Z+1  //POINTER TO PREC CONSTR LINE
72          MAT:=POINT+4
73          POINT:=MAT
74          RETURN  $)1
75
76  AND PAR8() BE  //DECIM REPRESENT OF 1ST CHAR OF ACT PARAM
77      $( IF \SCH(!(JO-1),!(MAT-4)-1) THEN
78           $( LET X=OUTPUT
79              SELECTOUTPUT(1)
80              WRITES("*NCONV8, NON-EXIST PARAM*N")
81              SELECTOUTPUT(X)
82              WRERR()
83              RETURN  $)
84         IF !(PASY+3)=0  RETURN  //EMPTY PARAM
85         DEC (!(!(PASY+2)))
86         RETURN  $)
87  .
88
89
90
91
```

```
GET "/VAR"

LET PARSY() BE      //SYMBOL GENERATION
$( LET Z=!(MAT-4)-2
     IF SCH(!(JO+1),Z) THEN
     $( DEC(!(PASY+2))  //CONVERT ALREADY EXISTING SYGEN IN
        JO+:=2
        RETURN $)     //CONSTR LINE
     $( LET L=POINT+2  //INSERT SYGEN IN GEN SYMBOLS STRING
        UNTIL L>MAT-1 DO $( !L:=!(L-3)
                            L-:=1 $)
     !(MAT-2):=SYGEN
     !(MAT-3):=!(JO+1)
     !(MAT-4):=Z
     !Z:=MAT-4
     POINT+:=3  //WRITE GEN SYMB IN CONSTR LINE
     DEC'SYGEN)
     MAT+:=3
     SYGEN+:=1
     JO+:=2    //NEXT CHAR OF CODE BODY
     RETURN $)2

AND PITER() BE  //ITERATIONS
   $(1 IF !(JO-1)='F' GOTO FU7
       IF POINT=MAT  $( JO+:=2 : RETURN $)
       CR-:=9   //INITIALIZE CONTEXT-CONTROLED ITERATION
       !CR:=MAT  //SAVE ADDR OF CONSTR LINE
       !(CR-1):=MAT  //PART OF CONSTR LINE SAVED FOR NEXT ITERATION
       !(CR-2):=POINT-1
       !(CR-7):=JO  //ADDR 1ST LINE OF ITERATION
       !(CR-8):=!(!(MAT-4)-1) //SAVE POINTER TO 1ST PAR (IF CONV 6!)
       TEST SCH(!(JO-1),!(MAT-4)-1)
       THEN $( !(CR-3):=TRUE    //PARAM ALREADY EXISTING
               !(CR-4):=!(PASY+1)  //SAVE PARAM
               !(CR-5),!(CR-6):=!(PASY+2),!(PASY+3)
               !POINT:=127
               !(POINT-1):=!(MAT-4)
               POINT+:=5
               MAT:=POINT $)
       OR $( !(CR-3):=FALSE   //NEW TEMPORARY PARAM
             !(CR-4):=!(JO-1)
             !POINT:=127
             !(POINT+1):=!(!(MAT-4)-1) //UPDATE POINTERS OF PARAM LIST
             !(!(MAT-4)-1):=POINT+1
             !(POINT+2):=!(JO-1)
             !(POINT+5):=!(MAT-4)
             POINT+:=9
             MAT:=POINT  $)
  RECH:  SCH(!(JO-1),!(MAT-4)-1)
         !(PASY+2):=!(CR-1)  //POINTER TO 1ST CHAR.OF ACT PARAM
         $(2 LET I,J=JO+1,!(CR-1)
             UNTIL !(JO+1)='*N' DO JO+:=1 //BREAK CHARS FROM I TO JO
             WHILE J<=!(CR-2) DO
         $(3 IF I=JO+1 THEN  //NO BREAK CHAR
             $( !(PASY+3):=!(CR-1) //ACT PARAM = CURRENT CHAR
```

```
56                  !(CR-1)+:=1
57                  JO+:=2
58                  RETURN  S)
59      ZAZ: IF !J=LPA THEN
60             S(4 LET G.PC=J.1  //SEARCH BALANCED SUBSTRING
61                 S(X J+:=1
62                      IF J>!(CR-2) GOTO PA1
63                      TEST !J\=RPA
64                      THEN S( IF !J=LPA  PC+:=1
65                              LOOP S)
66                      OR S( PC-:=1
67                            IF PC<0 GOTO PA1
68                            IF PC=0 S( J+:=1 ; GOTO ZAZ S)
69                            LOOP S)
70                  PA1: !(PASY+3):=G //NO BALANCED SUBSTRING : LPA BECOMES
71                      !(CR-1):=G+1                    //ACT PARAM
72                      JO+:=2
73                      RETURN  S)X REPEAT  S)4
74             FOR K=1 TO JO DO
75                 S( IF !J=!K THEN  //J POINTS TO A BREAK CHAR
76                       S( TEST J=!(CR-1)   //2D POINTER
77                          THEN !(PASY+3):=0  //EMPTY PARAM
78                          OR !(PASY+3):=J-1
79                          !(CR-1):=J+1  //1ST CHAR OF CONSTR LINE FOR
80                          JO+:=2                //NEXT ITERATION
81                          RETURN  S)  S)
82             J+:=1  S)3
83               !(PASY+3):=J-1  //END OF ITER BY EXHAUSTION
84               !(CR-1):=J
85               JO+:=2
86               RETURN  S)2
87
88  FU7: IF !JO=ZERO+8 GOTO FU8
89       IF MAT=POINT  S( JO+:=2   //INITIALIZE COUNT-CONTROLLED ITERATION
90                       RETURN  S)  //EMPTY CONSTR LINE
91       IF \ARITH(MAT.POINT-1)  THEN  S( JO+:=2 : RETURN S)
92       CR-:=5
93       !(CR-1):=!POINT  //NUMBER OF ITERATIONS TO EXECUTE
94       !CR:=0
95       JO+:=2
96       !(CR-2):=JO  //ADDR 1ST LINE OF ITERATION
97       POINT:=MAT
98       RETURN
99
100 FU8: IF CR>5VE  THEN S( JO+:=2 : RETURN S)
101      TEST !CR=0        //END OF ITERATION?
102      THEN S( TEST !(CR-1)<=1  //COUNT-CONTROLLED
103              THEN  S( CR+:=9 : JO+:=2 : RETURN S)  //END OF ITER
104              OR S( JO:=!(CR-7)
105                    !(CR-1)-:=1   //START NEXT ITERATION
106                    RETURN  S) S)
107      OR S( TEST !(CR-1)>!(CR-2)  //CONTEXT-CONTROLLED
108            THEN S( SCH(!(CR-4),!(MAT-4)-1)  //END
109                    IF !(CR-3)=TRUE  THEN
110                       S( !(PASY-2):=!(CR-5) //RESTORE OLD PARAM
```

```
111                          !(PASY+3):=!(CR-6) S)
112                    !!!(MAT-4)-1):=!(CR-8) //AND RESTORE OLD SITUATION
113                    MAT:=!CR
114                    POINT:=MAT
115                    CR-:=9
116                    JO+:=2
117                    RETURN S)
118              OR S( JO:=!(CR-7)  //START NEXT ITERATION
119                    GOTO RECH S)1
120
121
122
123
```

```
  1  GET "//VAR"
  2
  3  LET FUN1() BE
  4
  5  S(3 IF POINT=MAT/\!(JO+1)="*N"/\!(JO+2)=TELE THEN
  6  //EMPTY CONSTR LINE AND END OF CODE BODY
  7     S(4 LET X=OUTPUT
  8         SELECTOUTPUT(1)
  9         WRITEF("*NEMPTY CONSTR LINE,END CODE BODY:F1*N")
 10         SELECTOUTPUT(X)
 11         WRERR()
 12         JO+:=2
 13         RETURN    S)4
 14  TEST !(JO-2)\="*N"/\!(JO-3)\=127
 15  //OUTPUT LINE WITHOUT ATTEMPTING TO REMATCH IT
 16  THEN S(4 !POINT:=127
 17           WRILI()
 18           JO+:=2    // 2  TO AVOID SPURIOUS EMPTY LINE
 19           POINT:=MAT
 20           RETURN    S)4
 21  //NEXT LINE OF CODE BODY TAKEN AS FIELD DEFINITION
 22  OR S(4 JO+:=2
 23         POINT:=MAT
 24  CCC:   IF !JO="*N" THEN
 25            S(5 !POINT:=127   //OUTPUT LINE DIRECTLY
 26                WRILI()
 27                JO+:=1
 28                POINT:=MAT
 29                RETURN  S)5
 30         TEST !JO>=ZERO+1/\!JO<=ZERO+9
 31         THEN S(5 LET I=!JO  //PARAM FIELD
 32                  TEST \SCH(I,!(MAT-4)-1)
 33                  THEN S( LET X=OUTPUT  //NO ACT PAR
 34                          SELECTOUTPUT(1)
 35                          WRITEF("*NUNDEF PAR IN FIELD.FUN1*N")
 36                          SELECTOUTPUT(X)
 37                          WRERR()
 38                          WHILE !JO=I DO JO+:=1
 39                          GOTO CCC S)
 40                  OR
 41               S(7 LET J,K=!(PASY+1)+1,!(PASY+2)
 42                S(8 TEST K=J\/J=1    //IF J=1, EMPTY PARAM
 43                    THEN S(9 WHILE !JO=I DO //PAR<FIELD
 44                                S( !POINT:=PADD
 45                                   JO+:=1
 46                                   POINT+:=1 S)
 47                             GOTO CCC S)9
 48                    OR S(9 IF !JO=I DO
 49                                S( !POINT:=!K
 50                                   POINT+:=1 : JO+:=1 : K+:=1
 51                                   LOOP S)
 52                             GOTO CCC S)8 REPEAT S)5 //PAR>FIELD:TRUNCATE
 53         OR S(5 !POINT:=!JO  //NOT FIELD
 54                POINT+:=1 : JO+:=1
 55                GOTO CCC S)3
```

```
56
57  AND FUN4() BE //UNCOND SKIP
58    S( IF \SCH(49.!(MAT-4)-1) THEN
59         S( LET X=OUTPUT
60            SELECTOUTPUT(1)
61            WRITES("*NPAR4. NON-EXIST PARAM*N")
62            SELECTOUTPUT(X)
63            WRERR()
64            JO+:=2
65            RETURN  S)
66       IF !(!(PASY+3))=0 THEN  S( JO+:=2 : RETURN  S) //EMPTY PARAM
67       TEST ARITH(!(PASY+2).!(PASY+3))
68       THEN S( SK:=!POINT
69               JO+:=2
70               IF SKIP()   POINT:=MAT  S)
71       OR JO+:=2
72       RETURN  S)
73
74  AND FUN5() BE //SKIP ON STRING EQUALITY
75     S(1  IF \SCH(49.!(MAT-4)-1) GOTO RET
76        S(2 LET W=PASY
77            IF \SCH(50.!(MAT-4)-1) GOTO RET
78            TEST !(W+3)=0
79            THEN S( IF !(PASY+3)=0 GOTO KO
80                    GOTO K1 S)
81            OR S( IF !(PASY+3)=0 GOTO K1
82                  IF !(W+3)-!(W+2)\=!(PASY+3)-!(PASY+2) GOTO K1 S)
83            FOR J=0 TO !(PASY+3)-!(PASY+2)  DO
84                S( IF !(!(W+2)+J)\=!(!(PASY+2)+J) GOTO K1 S)?
85      KO:   IF !JO=ZERO GOTO SKI
86            JO+:=2
87            RETURN
88      K1:   IF !JO\=ZERO-1  S( JO+:=2 : RETURN  S)
89      SKI:  IF \SCH(51.!(MAT-4)-1) GOTO RET
90            IF !(!(PASY+3))=0  S( JO+:=2 : RETURN  S) //EMPTY VALUE
91            TEST ARITH(!(PASY+2).!(PASY+3))
92            THEN  S( SK:=!POINT
93                     JO+:=2
94                     IF SKIP()  POINT:=MAT  S)
95            OR JO+:=2
96            RETURN
97      RET:  S( LET X=OUTPUT
98               SELECTOUTPUT(1)
99               WRITES("*NFUN5. PARAM NON-EXIST*N")
100              SELECTOUTPUT(X)
101              WRERR()
102              JO+:=2
103              RETURN   S)1
104
105 AND FUN6() BE  //SKIP ON RELAT VALUE OF 2 A.E.
106    S(1 IF \SCH(49.!(MAT-4)-1) GOTO ER
107        IF !(!(PASY+3))=0 GOTO ER
108        IF \ARITH(!(PASY+2).!(PASY+3)) THEN
109                        S( POINT+:=1 : GOTO RE S)
110        POINT+:=1
```

```
111         IF \SCH(50,!(MAT-4)-1) THEN S( POINT-:=1 : GOTO ER S)
112         IF !(!(PASY+3))=0 S( POINT-:=1 : GOTO ER S)
113         IF \ARITH(!(PASY+2),!(PASY+3)) GOTO RE
114         IF !JO=MINU THEN
115               S( IF !(POINT-1)<!POINT GOTO SKI
116                  GOTO RE S)
117         IF !JO=ZERO THEN
118               S( IF !(POINT-1)=!POINT GOTO SKI
119                  GOTO RE S)
120         IF !JO=ZERO+1 THEN
121               S( IF !(POINT-1)\=!POINT GOTO SKI
122                  GOTO RE S)
123         IF !JO=PLUS THEN
124               S( IF !(POINT-1)>!POINT GOTO SKI S)
125    RE:  POINT-:=1
126         JO+:=2
127         RETURN
128    SKI: POINT-:=1
129         IF \SCH(51,!(MAT-4)-1) GOTO ER
130         IF !(!(PASY+3))=0 S( JO+:=2 : RETURN S)
131         TEST ARITH(!(PASY+2),!(PASY+3))
132         THEN S( SK:=!POINT
133                 JO+:=2
134                 IF SKIP() THEN POINT:=MAT S)
135         OR JO+:=4
136         RETURN
137    ER:  S( LET X = OUTPUT
138            SELECTOUTPUT(1)
139            WRITES("*NFUN6. PAR EMPTY OR NON-EXIST*N")
140            SELECTOUTPUT(X)
141            WRERR()
142            JO+:=2
143            RETURN  S)1
144
145
146
147
```

```
GET "/VAR"
//INPUT FROM FILE /DATAST :  OUTPUT ON FILE /RESULT

LET READFLAG() BE
// INPUT FLAG LINE
//NEWLINES BEFORE FLAG LINE ARE IGNORED
    S(   SELECTINPUT(INFI)
         SELF:=RDCH() REPEATWHILE SELF='*N'
         SPAF:=RDCH(): TELF:=RDCH(): ESC:=RDCH(): ZERO:=RDCH()
         PADD:=RDCH(): LPA:=RDCH(): PLUS:=RDCH(): MINU:=RDCH()
         MUL:=RDCH(): DIV:=RDCH(): RPA:=RDCH()  //IF USER DEFINES
          //ZERO \= '0', INTERN CODE OF DIGITS 1 TO 9 MUST FOLLOW
         AAA: UNLESS RDCH()='*N' GOTO AAA    //NEW DEFINED ZERO
         RETURN S)

AND READING() BE
//READING OF INPUT LINE
      S(  SELECTINPUT(INFI)
          S(  !POINT:=RDCH():  POINT+:=1  S)
               REPEATUNTIL !(POINT-1)='*N'\/!(POINT-1)=SELF
          !(POINT-1):=127
          RETURN     S)

AND WRERR() BE
//ERROR MESSAGES
   S( LET K.X=MAT.OUTPUT
      SELECTOUTPUT(1)
      WRITEF("CONSTR LINE: ")
      UNTIL K=POINT DO S( WRCH(!K) : K+:=1 S)
      WRCH('*N')
      K:=!(MAT-4)
      UNTIL K=CONST+2 DO
          S(2 WRITEF("CONSTR LINE: ")
              S(3 LET P=K
                  UNTIL !P=127 DO S( WRCH(!P) : P+:=1 S)
                  WRCH('*N')
                  K:=!(K-4)  S)2
      WRITEF("INPUT LINE: ")
      UNTIL !K=127 DO S( WRCH(!K) : K+:=1 S)
      WRCH('*N'):WRCH('*N')
      SELECTOUTPUT(X)
      RETURN   S)

AND WRILI() BE
//WRITING OF AN OUTPUT LINE
      S(  LET K.X=MAT.OUTPUT
          SELECTOUTPUT(OUTFI)
          UNTIL !K=127 DO S( WRCH(!K): K+:=1 S)
          WRCH('*N')
          SELECTOUTPUT(X)
          RETURN    S)
```

TEXTE SOURCE 1

TEXTE DE DEFINITION

```
TYPE /DATAST
$'$'0 (+-*/)
END$
      END'F1$
'F0$
$
' EQU '$
'F3$
$
SKIP '$
'F4$
$
IF ' = ' SKIP '$
'F50$
$
IF ' NE ' SKIP '$
'F51$
$
VAR ',1$
'F1$
11111 RESERVE  2222222222$
$
DO ' ' ' ' SKIP '$
IF '41 = '20 SKIP 2$
IF '41 = '30 SKIP 4$
      FETCH    '20'F1$
'F1$
      11111111 3333333333333333$
SKIP '50$
$
SAVE '$
IF '10 = AC SKIP 1$
      STORE    '10'F1$
AC EQU '10$
$
'='+'$
DO ADD '20 '30 AC SKIP 1$
      ADD      '20'F1$
SAVE '10$
$
'='-'$
DO SUB '20 '30 AC SKIP 2$
      NEGATE'F1$
      ADD      '20'F1$
SAVE '10$
$
'='*'$
DO MULT '20 '30 AC SKIP 1$
      MULT     '20'F1$
SAVE '10$
$
'='/'$
DO DIV '20 '30 AC SKIP 2$
      FETCH    '20'F1$
      DIV      '30'F1$
SAVE '10$
$$
```

TEXTE DE GENERATION

```
A=B+C$
DOG=EAST+A$
MITTS=DOG/EAST$
AC=EAST*MITTS$
AC=B-AC$
NO=MITTS-EAST$
NO=NO/A$
VAR A,1$
VAR B,1$
VAR C,1$
VAR DOG,1$
VAR EAST,1$
VAR MITTS,1$
VAR NO,1$
END$
```

TEXTE CIBLE 1

```
WHAT NOW?
TYPE /RESULT
      FETCH    B
      ADD      C
      STORE    A
      ADD      EAST
      STORE    DOG
      DIV      EAST
      STORE    MITTS
      MULT     EAST
      NEGATE
      ADD      B
      FETCH    MITTS
      SUB      EAST
      STORE    NO
      DIV      A
      STORE    NO
A     RESERVE  1
B     RESERVE  1
C     RESERVE  1
DOG   RESERVE  1
EAST  RESERVE  1
MITTS RESERVE  1
NO    RESERVE  1
      END
```

TEXTE SOURCE 2

TEXTE DE DEFINITION

```
TYPE /DATAST
$'$'0 (+-*/)
END$
VARS  RESERVE  '00$
      END'F1$
'F0$
$
' EQU '$
'F3$
$
SKIP '$
'F4$
$
IF ' = ' SKIP '$
'F50$
$
IF ' NE ' SKIP '$
'F51$
$
VAR ',$
'F1$
11111 RESERVE  2222222222$
$
DO ' ' ' ' SKIP '$
IF '41 = '20 SKIP 2$
IF '41 = '30 SKIP 5$
      FETCH    VARS+'22'F1$
VARS+'32'36$
'F1$
      11111111 3333333333333333$
SKIP '50$
$
SAVE '$
IF '10 = AC SKIP 1$
      STORE    VARS+'12'F1$
AC EQU '10$
$
'='+'$
DO ADD '20 '30 AC SKIP 1$
      ADD      VARS+'22'F1$
SAVE '10$
$
'='-'$
DO SUB '20 '30 AC SKIP 2$
      NEGATE'F1$
      ADD      VARS+'22'F1$
SAVE '10$
$
'='*'$
DO MULT '20 '30 AC SKIP 1$
      MULT     VARS+'22'F1$
SAVE '10$
$
'='/'$
DO DIV '20 '30 AC SKIP 2$
      FETCH    VARS+'22'F1$
      DIV      VARS+'22'F1$
SAVE '10$
$$
A=B+C$
DOG=EAST-A$
MITTS=DOG/EAST$
AC=EAST*MITTS$
AC=B-AC$
NO=MITTS-EAST$
NO=NO/A$
END$
```

TEXTE DE GENERATION

```
WHAT NOW?
C
WHAT NOW?
/STAGE2

LIMPID  1.1
BCPL
WHAT NOW?
TYPE /RESULT
      FETCH    VARS+1
      ADD      VARS+2
      STORE    VARS+3
      NEGATE
      ADD      VARS+4
      STORE    VARS+5
      DIV      VARS+4
      STORE    VARS+6
      MULT     VARS+4
      NEGATE
      ADD      VARS+1
      FETCH    VARS+6
      SUB      VARS+4
      STORE    VARS+7
      DIV      VARS+3
      STORE    VARS+7
VARS  RESERVE  8
      END
```

TEXTE CIBLE 2

TEXTE DE DEFINITION

```
 1  S'S'0 (+-*/)
 2  ' = ABS(')S
 3        FETCH     '20'F1S
 4        PLUSJUMP  C'00'F1S
 5        NEGATE'F1S
 6  '00'96S
 7  'F1S
 8  C999 STORE     1111111111S
 9  S
10  LENG ' ' ' ' 'S
11  '15 '25 '35 '45 '55S
12  S
13  DEC ' ' ' ' 'S
14  '18 '28 '38 '48 '58S
15  S
16  VAL 'S
17  '12'F1S
18  S
19  COPY TO 'S
20  'F21S
21  S
22  DELETE TO 'S
23  'F20S
24  S
25  ENDS
26  VARS  RESERVE  286'F1S
27  IF MAX LT 0 SKIP 2S
28  MAX+1'96S
29  TEMP  RESERVE  '94'F1S
30        END'F1S
31  'F0S
32  S
33  ' PRI (')S
34  IF '10 = FETCH SKIP 5S
35  IF CNT LT MAX SKIP 1S
36  MAX SET MAX+1S
37  CNT SET CNT+1S
38  CNT'80S
39        STORE     TEMP+'81'F1S
40  UP'96S
41  UP EQU 0S
42  '20'27+-S
43  IF '91 = + SKIP 3S
44  IF '91 = - SKIP 4S
45  FETCH PRI '20S
46  SKIP 3S
47  ADD PRI '20S
48  SKIP 1S
49  SUB PRI '20S
50  UP EQU '23S
51  'F8S
52  IF '10 = FETCH SKIP 4S
53  IF '10 = ADD SKIP 1S
54        NEGATE'F1S
55        ADD       TEMP+'81'F1S
```

```
 56  CNT SET CNT-1$
 57  $
 58  IF ' LT ' SKIP '$
 59  'F6-$
 60  $
 61  IF ' LE ' SKIP '$
 62  IF '10 LT '20 SKIP '30+1$
 63  'F60$
 64  $
 65  IF ' EQ ' SKIP '$
 66  'F60$
 67  $
 68  IF ' NEQ ' SKIP '$
 69  'F61$
 70  $
 71  IF ' GE ' SKIP '$
 72  IF '10 GT '20 SKIP '30+1$
 73  'F60$
 74  $
 75  IF ' GT ' SKIP '$
 76  'F6+$
 77  $
 78  ' PRI '$
 79  '20'9701234567895
 80  IF '90 NE  SKIP 3$
 81  'F1$
 82        11111111 =22222222$
 83  'F9$
 84  '21'26
 85  'F1$
 86        11111111 VARS+222$
 87  'F8$
 88  $
 89  '='$    TRANSLATION OF ASSIGNEMENT STMT
 90  OLD'96$                 MUST ACCESS OLD FROM MEMORY
 91  OLD EQU $               SET PREVIOUS OP
 92  '20'27+-$               BREAK A.F. ON OPERATORS + AND -
 93  IF '91 = + SKIP 5$
 94  IF '91 = - SKIP 2$      DECODE PREVIOUS OP
 95  FETCH PRI '20$
 96  SKIP 3$                 GO SET NEXT OP
 97  SUB PRI '20$
 98  SKIP 1$
 99  ADD PRI '20$
100  OLD EQU '23$            SET NEXT OPERATOR
101  'F8$                    ADVANCE TO NEXT OPERAND
102  STORE PRI '10$
103  $
104  ' EQU '$
105  'F3$
106  $
107  SKIP '$
108  'F4$
109  $
110  IF ' = ' SKIP '$
```

```
111  'F50$
112  $
113  IF ' NE ' SKIP '$
114  'F51$
115  $
116  ' SET '$
117  '10 EQU '24$
118  $
119  VARS$
120  VCT'95$
121  VCT EQU 0$
122  ABCDEFGHIJKLMNOPQRSTUVWXYZ'17$
123  '10 EQU '91$
124  VCT SET VCT+1$
125  0123456789'27$
126  '10'20 EQU '91$
127  VCT SET VCT+1$
128  'F8$
129  'F8$
130  MAX EQU -1$
131  CNT EQU -1$
132  $
133  OUTPUT ' ' '$
134  IF '10 GE '20 SKIP '30$
135  'F7$
136  AVX
137  'F8$
138  $
139  RECURSION '$
140  '10'26$
141  '20'36$
142  '30'46$
143  '40'56$
144  '50'66$
145  '60'76$
146  '70'86$
147  '80'96$
148  RECURSION '10$
149  $
150  DATA  '$
151  DATA  '26$
152  '10'17012345678$
153  IF '10 = /'' SKIP 3$
154  IF '10 = ''/'' SKIP 2$
155  IF '10 = ''.'' SKIP 3$
156  IF '10 = ''/ SKIP 5$
157  '20/14'13'26$
158  SKIP 1$
159  '20.14'13'26$
160  'F7$
161  'F8$
162  '20/'F1$
163  'F8$
164  $$
165  A = ABS(B)$
```

TEXTE DE GENERATION

```
166  LENG 3 ABCDEFGHIJKLMNOPQRSTUVWXYZ -1234 1488 !?/?!$
167  OUTPUT 10 5 5$
168  OVER THIS
169  NOT THIS
170  PRINT NOW
171  RECURSION WITHOUT ENDTESTS
172  COPY TO TOTS
173  ABCD
174  PRINTS
175  LIST
176  TOTALS
177  ZOZOS
178  DELETE TO TOTS
179  FIRST
180  SECOND
181  TOTALITY
182  D = ABS(E)$
183  TOTS
184  X = ABS(Y)$
185  DEC 0 & AD ... )OS$
186  X = ABS(Y)$
187  IF 1 NE 2 SKIP 2$
188  HERE
189  THERE
190  PRINT
191  IF 2 LE 2 SKIP 1$
192  NO PRINT
193  TYPE
194  COPY TO REPS
195  THIS COULD
196  BE FORTRAN
197  REPLACE
198  DATA  /'1','2'/'6'/'3','7','8'/'4'/
199  A EQU 123$
200  B EQU 2XY$
201  X = ABS(R)$
202  XAZ EQU -435
203  WE EQU ABCDEFGHIJ$
204  VAL A$
205  VAL XAZ$
206  VAL RJ$
207  VAL B$
208  VAL XA$
209  VAL WE$
210  B EQU -12$
211  A EQU 48
212  WE EQU 2$
213  VAL W$$
214  WE EQU -12345$
215  VAL RJ$
216  VAL WE$
217  VAL XA$
218  VAL A$
219  VAL B$
220  VAL A0$
```

```
221  VAL S6S
222  VAL Z9S
223  VAL T5S
224  VAL Q4S
225  GOOD LUCK
226  VARSS
227  Z9=S6-(D5-(W4-(123+345)-G8+R4)+(S3-T5))-(E5-Q4)$
228  END
229
230
231
232
```

```
      FETCH     B
      PLUSJUMP  C1
      NEGATE
C1    STORE     A
1 26 5 4 5
PRINT NOW
ABCD
PRINT
LIST
2020
      FETCH     E
      PLUSJUMP  C2
      NEGATE
C2    STORE     D
TOT
      FETCH     Y
      PLUSJUMP  C3
      NEGATE
C3    STORE     X
48 64 65 44 41
      FETCH     Y
      PLUSJUMP  C4
      NEGATE
C4    STORE     X
PRINT
TYPE
THIS COULD
BE FORTRAN
DATA  /1H1,1H2/1H5/1H3,1H7,1H8/1H4/
      FETCH     R
      PLUSJUMP  C5
      NEGATE
C5    STORE     X
123
-435
6
@XY
7
ABCDEFGHIJ
2
6
-12345
7
48
-12
8
9
10
11
12
GOOD LUCK
       FETCH    VARS+205
       STORE    TEMP+0
       FETCH    VARS+39
       STORE    TEMP+1
```

```
56          FETCH     VARS+247
57          STORE     TEMP+2
58          FETCH     =123
59          ADD       =345
60          NEGATE
61          ADD       TEMP+2
62          SUB       VARS+75
63          ADD       VARS+192
64          NEGATE
65          ADD       TEMP+1
66          STORE     TEMP+1
67          FETCH     VARS+202
68          SUB       VARS+215
69          ADD       TEMP+1
70          NEGATE
71          ADD       TEMP+0
72          STORE     TEMP+0
73          FETCH     VARS+50
74          SUB       VARS+181
75          NEGATE
76          ADD       TEMP+0
77          STORE     VARS+285
78  VARS    RESERVE   286
79  TEMP    RESERVE   3
80          END
```

T A B L E D E S M A T I E R E S

ETUDE COMPARATIVE DE MACROGENERATEURS - ERRATA

(OK)

1.13 ligne 5 : lire "évaluée" au lieu de "évoluée"

1.15 ligne 23 : lire "seront conditionnées par" au lieu de "se déduiront immédiatement de"

1.16 ligne 9 : lire"le non-terminal" au lieu de "le nom terminal"

1.22 lignes 15 à 18 : lire

b.- soit une macro dont la syntaxe d'appel serait

$$A ::= \text{IF arg THEN arg} \left\{ \begin{array}{l} [\text{ELSE arg ?}] \\ [\text{ELSE A ?}] \end{array} \right\} \text{END}$$

où les arguments ne peuvent pas contenir les atomes 'IF', 'THEN' ou 'ELSE'

1.23 trois dernières lignes : lire

3.- l'écriture du texte de génération est souple. Outre les limitations commentées dans les pages 1.20 et 1.21, notons que l'utilisateur peut spécifier uniquement la syntaxe de ce qui suit le nom de la macro, jamais celle du texte qui précède. On remarqueraque la grammaire des structures de délimiteurs est régulière, ce qui est assez restrictif.

1.24 lignes 5 et 6 : lire "de chaque macrodéfinition" au lieu de "des macrodéfinitions"

1.25 lignes 11 à 14 : lire

$$\left\{ \begin{array}{ll} e_{i,0}\ ' e_{i,2}\ ' \ldots\ldots\ldots ' e_{i,2n_i}\ ' [\, e_{i,2(n_i+1)}\ ?\,] [.\,?] \ \underline{NL} & (\text{I}) \\ ' e_{i,1}\ ' e_{i,2}\ ' \ldots\ldots\ldots ' e_{i,2n_i-1}\ ' [\, e_{i,2n_i+1}\ ?\,] [\,.\,?] \underline{NL} & (\text{II}) \end{array} \right.$$

$$\text{avec} \left\{ \begin{array}{l} 1 \leqslant n_i \leqslant 8 \\ e_{i,j} = \underline{\text{chaîne}} \qquad j \quad 0,1,\ldots\ldots,18 \end{array} \right.$$

ligne 18 : lire "$s_0 s_1 s_2 \ldots\ldots s_k$"

ligne 21 : lire "s_j = string" au lieu de "s_k = string"

1.27 ligne 4 : lire "qu'il y a de nombres d'arguments possibles"

2.15 ajouter en bas de page :

Note : pour les besoins de la cause, nous n'avons pas utilisé l'instruction-assemb
LPR dans l'exemple

2.20 ligne 26 : lire "STORE X" au lieu de "LOAD X"

2.32 ligne 7 : lire "&(AD,1" au lieu de "&(AD.1"

2.39 ligne 23 : lire "I RES 1" au lieu de "i RES 1"

A.2 ajouter en bas de page :

Note : la fonction-système 'eFO' indique la fin de macrogénération